XZ130

ACQIRIS PROGRAMMER'S GUIDE

December 2004

The information in this document is subject to change without notice and may not be construed as in any way as a commitment by Acqiris. While Acqiris makes every effort to ensure the accuracy and contents of the document it assumes no responsibility for any errors that may appear.

All software described in the document is furnished under license. The software may only be used and copied in accordance with the terms of license. Instrumentation firmware is thoroughly tested and thought to be functional but it is supplied "as is" with no warranty for specified performance. No responsibility is assumed for the use or the reliability of software, firmware or any equipment that is not supplied by Acqiris SA or its affiliated companies.

Any versions of this manual which are supplied with a purchased product will be replaced at your request with the latest revision in electronic format. At Acqiris we appreciate and encourage customer input. If you have a suggestion related to the content of this manual or the presentation of information, please contact your local Acqiris representative or Acqiris Technical Support (support@acqiris.com) or come visit our web site at http://www.acqiris.com.

Trademarks: product and company names listed are trademarks or trade names of their respective companies

Acqiris Headquarters:

Acqiris SA
18, chemin des Aulx
CH-1228 Plan-les-Ouates
Geneva
Switzerland

Tel: +41 22 884 33 90
Fax: +41 22 884 33 99

Acqiris USA:

Acqiris LLC
234 Cromwell Hill Rd.
P.O. Box 2203
Monroe, NY 10950-1430
USA

Tel: 845 782 6544
Fax: 845 782 4745

Acqiris Asia-Pacific:

Acqiris Pty Ltd
Suite 7, Level 1
407 Canterbury Road,
P.O. Box 13
Surrey Hills 3127
Australia

Tel: +61 3 9888 4586
Fax: +61 3 9849 0861

© Copyright December 2004, Acqiris SA. All rights reserved.

CONTENTS

1. INTRODUCTION 6
1.1. Message to the User 6
1.2. Using this Manual 6
1.3. Conventions Used in This Manual 7
1.4. Warning Regarding Medical Use 7
1.5. Warranty 7
1.6. Warranty and Repair Return Procedure, Assistance and Support 7
1.7. System Requirements 7

2. PROGRAMMING ENVIRONMENTS & GETTING STARTED 8
2.1. Visual C++ 8
2.2. LabWindows/CVI 8
2.3. LabVIEW 8
2.3.1. Getting Started VI 9
2.3.2. Example Scope VI 9
2.3.3. Accumulated Waveform Example VI 10
2.4. Visual Basic 10
2.5. MATLAB 11
2.5.1. Examples 11
2.5.2. Setup and Settings 12
2.5.3. Process an acquisition 13
2.6. Phar Lap ETS 15
2.7. VxWorks (Tornado) 15
2.7.1. Compiling 15
2.7.2. Loading 15
2.7.3. Running the Program 16
2.8. Linux 16

3. PROGRAMMING AN ACQIRIS DIGITIZER 17
3.1. Programming Hints 17
3.2. Device Initialization 17
3.2.1. Identification by Order Found 18
3.2.2. Identification by Serial Number 18
3.2.3. Identification by Bus/Slot Number 18
3.2.4. Firmware initialization (AP-FAMILY/12-bit-FAMILY) 19
3.2.5. Automatic Definition of MultiInstruments 19
3.2.6. Manual Definition of MultiInstruments 20
3.2.7. AqGeo.map file positioning 21
3.2.8. Simulated Devices 22
3.2.9. Terminating an Application 22
3.3. Device Configuration 22
3.4. Configuring Averagers 24
3.4.1. Basic configuration 24
3.4.2. Dithering 25
3.4.3. 'Fixed Pattern' Background Subtraction 25
3.4.4. Configuring Noise Suppressed Accumulation (NSA) 27
3.5. Configuring SSR Analyzers 27
3.5.1. Acquisition Parameters 27
3.5.2. Readout configuration 28
3.5.3. Time stamps 29
3.6. Configuring AP101/AP201 Analyzers 29
3.7. Data Acquisition 30
3.7.1. Starting an Acquisition 31
3.7.2. Checking if Ready for Trigger 31
3.7.3. Waiting for End of Acquisition 31

3.7.4. Stopping/Forcing an Acquisition 32
3.7.5. Analyzer Autoswitch mode 32
3.8. Data Readout 34
3.8.1. Reading Digitizer Waveforms with the Universal Read Function 34
3.8.2. Reading Sequences of Waveforms 35
3.8.3. Averaging Waveforms in a Digitizer 37
3.8.4. Reading an Averaged Waveform from an Averager 37
3.8.5. Reading a RT Add/Subtract Averaged Waveform from an Averager 39
3.8.6. Reading SSR Analyzer Waveforms 39
3.8.7. Reading AP101/AP201 Analyzer Waveforms 40
3.9. Trigger Delay and Horizontal Waveform Position 46
3.10. Horizontal Parameters in Acquired Waveforms 46
3.11. Sequence Timestamps 47
3.12. External Clock and Reference 48
3.12.1. External Reference 48
3.12.2. External Clock (Continuous) 49
3.12.3. External Clock (Start/Stop) 50
3.13. ASBus Operation 52
3.13.1. Channel Numbering with ASBus 52
3.13.2. Trigger Source Numbering with ASBus 52
3.14. Special Operating Modes 54
3.14.1. Frequency Counter 54
3.14.2. 'Start on Trigger' 55
3.14.3. 'Sequence Wrap' 55
3.15. Readout of Battery Backed-up Memories 56
3.15.1. Preparations before Power-Off 56
3.15.2. Recovery after Power-Off 57
3.16. Reading the Digitizer Temperature 57

4. ATTRIBUTES 58
4.1. Attribute dependencies and coerced values 58
4.2. Functional grouping of attributes 59
4.3. Traditional API grouping of attributes 60
4.4. Detailed description of attributes (Alphabetically ordered) 62
axDitherRange 62
axNbrRoundRobins 62
axNbrWaveforms 63
axP1Signal 63
axP2Signal 64
axSamples 65
axSegments 65
axStartDelay 66
axStartVetoEnable 66
axStopDelay 67
axTrigAlways 67
axTrigResync 68
chAxInvertData 68
chAxNoiseBase 69
chAxNoiseBaseEnable 69
chAxTDCHistoEnable 70
chAxTDCHistoIncr 70
chAxTDCHistoMinTOT 71
chAxThreshold 71
chAxThresholdEnable 72
channelUse 72
chAttenuation 73
chBandwidthLimit 73

chCoupling ... 74
chFullScale ... 74
chImpedance ... 75
chInput ... 75
chOffset ... 76
chScale2ndFactor ... 76
chScaleFactor ... 77
chScaleOffset ... 77
chTime2ndOffset ... 78
chTime3rdOffset ... 78
chTimeOffset ... 79
chUnits ... 79
ckDelay ... 80
ckThreshLvl ... 80
ckType ... 81
convertersPerChan ... 81
ctrlIOASignal ... 82
ctrlIOBSignal ... 82
ctrlIOPxiRef10MHz ... 83
ctrlIOPxiStar ... 83
ctrlIOTrOutLevel ... 84
ctrlIOTrOutResync ... 84
extCkRatio ... 85
extInputFreq ... 85
extSamplingRate ... 86
extSparsing ... 86
input ... 87
name ... 87
samples ... 88
samplingFreq ... 88
samplingIntv ... 89
segments ... 89
timeWindow ... 90
trClass ... 90
trCoupling ... 91
trDelay ... 91
trRelThres1 ... 92
trRelThres2 ... 92
trSlope ... 93
trSrcEnabled ... 93
trThres1 ... 94
trThres2 ... 94
trThresUnit ... 95
trTVField ... 95
trTVLine ... 96
trTVStd ... 96

5. APPENDIX A: ESTIMATING DATA TRANSFER TIMES ... 97

5.1. Principles & Formulas ... 97
5.2. Examples ... 98
5.3. Comparison Chart for Typical Transfers ... 99

1. Introduction

1.1. Message to the User

Congratulations on having purchased an Acqiris data conversion product. Acqiris Digitizers are high-speed data acquisition modules designed for capturing high frequency electronic signals. To get the most out of the products we recommend that you read the accompanying product User Manual, this Programmer's Guide and the Programmer's Reference manual carefully. We trust that the product you have purchased as well as the accompanying software will meet with your expectations and provide you with a high quality solution to your data conversion applications.

1.2. Using this Manual

This guide assumes you are familiar with the operation of a personal computer (PC) running a Windows 95/98/2000/NT4/XP or other supported operating system. In addition you ought to be familiar with the fundamentals of the programming environment that you will be using to control your Acqiris product. It also assumes you have a basic understanding of the principles of data acquisition using either a waveform digitizer or a digital oscilloscope.

The **User Manual** that you also have received (or have access to) has important and detailed instructions concerning your Acqiris product. You should consult it first. You will find the following chapters there:

Chapter 1 ***OUT OF THE BOX,*** describes what to do when you first receive your new Acqiris product. Special attention should be paid to sections on safety, packaging and product handling. Before installing your product please ensure that your system configuration matches or exceeds the requirements specified.

Chapter 2 ***INSTALLATION***, covers all elements of installation and performance verification. Before attempting to use your Acqiris product for actual measurements we strongly recommend that you read all sections of this chapter.

Chapter 3 ***PRODUCT DESCRIPTION***, provides a full description of all the functional elements of your product.

Chapter 4 ***RUNNING THE ACQIRIS DEMONSTRATION APPLICATION***, describes either

the operation of AcqirisLive 2.13, an application that enables basic operation of Acqiris digitizers or averagers in a Windows 95/98/2000/NT4/XP environment;

the operation of AP_SSRDemo and in the following chapter APx01Demo, applications that enable basic operation of Acqiris analyzers in a Windows 95/98/2000/NT4/XP environment;

This Programmer's Guide is divided into 4 separate sections.

Chapter 1 ***INTRODUCTION,*** describes what can be found where in the documentation and how to use it.

Chapter 2 ***PROGRAMMING ENVIRONMENTS & GETTING STARTED***, provides a description for programming applications using a variety of software products and development environments.

Chapter 3 ***PROGRAMMING AN ACQIRIS DIGITIZER***, provides information on using the device driver functions to operate an Acqiris digitizer.

Chapter 4 ***ATTRIBUTES***, contains reference information about attributes. The attribute interface to the driver can be used with AcqirisMAQS, the MATLAB interface, and the SP201 Software Development Kit.

The accompanying **Programmer's Reference manual** is divided into 2 sections.

Chapter 1 **INTRODUCTION,** describes what can be found where in the documentation and how to use it.

Chapter 2 **DEVICE DRIVER FUNCTION REFERENCE**, contains a full device driver function reference. This documents the traditional Application Program Interface (API) as it can be used in the following environments:

LabWindowsCVI, Visual C++, LabVIEW, Visual Basic, Visual Basic .NET.

1.3. Conventions Used in This Manual

The following conventions are used in this manual:

This icon to the left of text warns that an important point must be observed.

WARNING Denotes a warning, which advises you of precautions to take to avoid being electrically shocked.

CAUTION Denotes a caution, which advises you of precautions to take to avoid electrical, mechanical, or operational damages.

NOTE Denotes a note, which alerts you to important information.

Italic text denotes a warning, caution, or note.

Bold Italic text is used to emphasize an important point in the text or a note

`mono` text is used for sections of code, programming examples and operating system commands.

Certain features are common to several different modules. For increased readability we have defined the following families:

DC271-FAMILY	DC135/DC140/DC211/DC211A/DC241/DC241A/ DC271/DC271A/DP214/DP235/DP240
AP-FAMILY	AP240/AP235/AP100/AP101/AP200/AP201
12-bit-FAMILY	DC440/DC438/DC436/DP310/DP308/DP306

1.4. Warning Regarding Medical Use

The Digitizer cards are not designed with components and testing procedures that would ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of these cards involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user. These cards are ***not*** intended to be a substitute for any form of established process or equipment used to monitor or safeguard human health and safety in medical treatment.

WARNING: ***The modules discussed in this manual have not been designed for making direct measurements on the human body. Users who connect an Acqiris module to a human body do so at their own risk.***

1.5. Warranty

Please refer to the appropriate User Manual.

1.6. Warranty and Repair Return Procedure, Assistance and Support

Please refer to the appropriate User Manual.

1.7. System Requirements

Please refer to the appropriate User Manual.

2. Programming Environments & Getting Started

Acqiris supplies sample programs as a starting point for the development of user-specific applications. For Windows systems there are samples for the Visual C/C++, Visual Basic, LabWindows/CVI, LabVIEW, and MATLAB. For Phar Lap ETS and VxWorks real-time systems there are sample programs for Visual Studio and Tornado, respectively. An Application Program Interface (API) hlp file, with a shortcut named Acqiris Digitizer Driver Help, is available and contains condensed descriptions of all of the interface functions.

Preliminary remark: *it is assumed in the following that the hardware and Acqiris software installations (see User Manual chapter 2) have already been completed.*

⚠ ***NOTE:*** *Visual C/C++, VxWorks, and LabWindows/CVI all rely on the standard VISA types defined by VXIplug&play Systems Alliance (VISA). The* ***visatype.h*** *include file can be found in the include directory created at installation.*

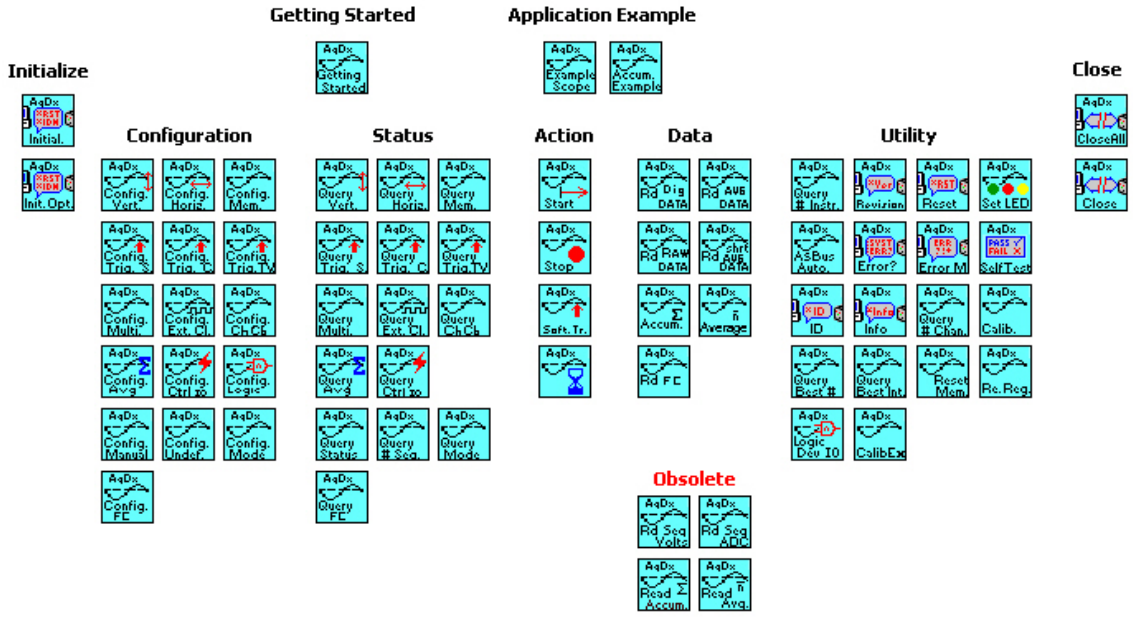
2.1. Visual C++

- Open either the project file **GetStartedVC.dsp** or **GetStartedAvgVC.dsp** in VisualC++ and build the project.
- Note that you should insert the line `#include "AcqirisD1Import.h"` at the beginning of every file that will access Acqiris Device Driver functions.
- The project should link to the **AqDrv4.lib** file.

2.2. LabWindows/CVI

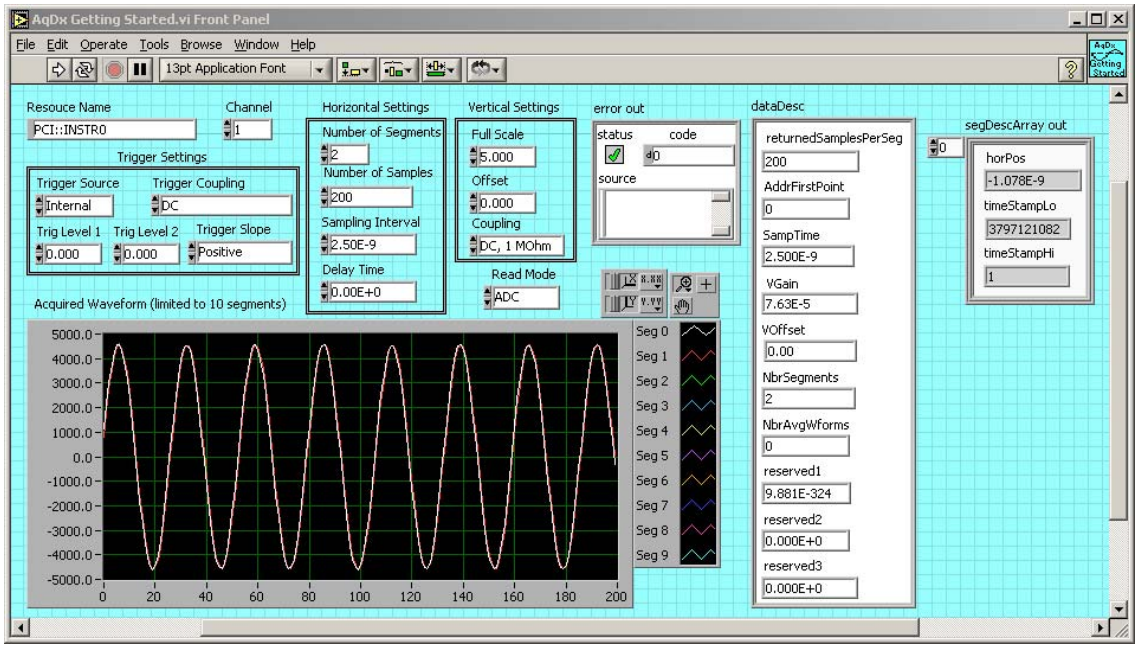
- Open the project file **GetStarted.prj** in LabWindows/CVI and build the project.
- Note that you should insert the line `#include "AcqirisD1Import.h"` at the beginning of every file that will access Acqiris Device Driver functions.
- The project should include the **AqDrv4.lib** file.

2.3. LabVIEW



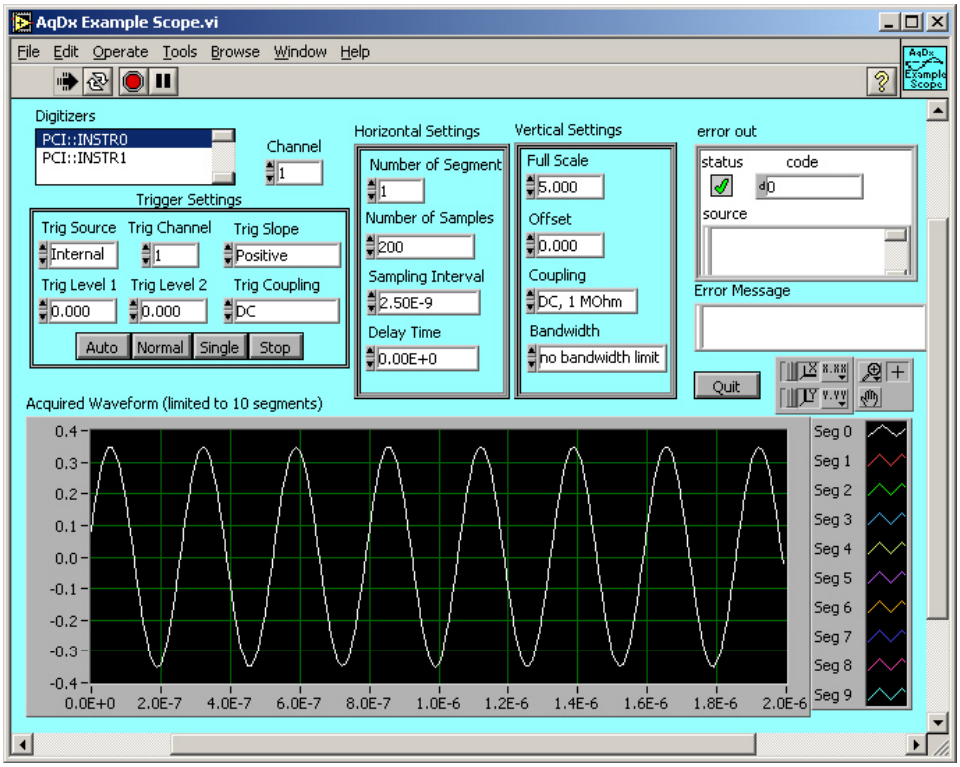
The AqDx (Acqiris Digitizer) driver for LabVIEW conforms to National Instruments' Instrument Driver Standard. If LabVIEW is installed on the target machine when the Acqiris software is installed, the AqDx driver will be copied to the LabVIEW n\Instr.lib directory. The driver function VI's can then be found on the Functions palette (block diagram), Instrument Drivers subpalette. There is also a Getting Started VI, as well as some example VI's. The standard API help file is available from within LabVIEW. The Revision Query VI gives information on the current version.

2.3.1. Getting Started VI



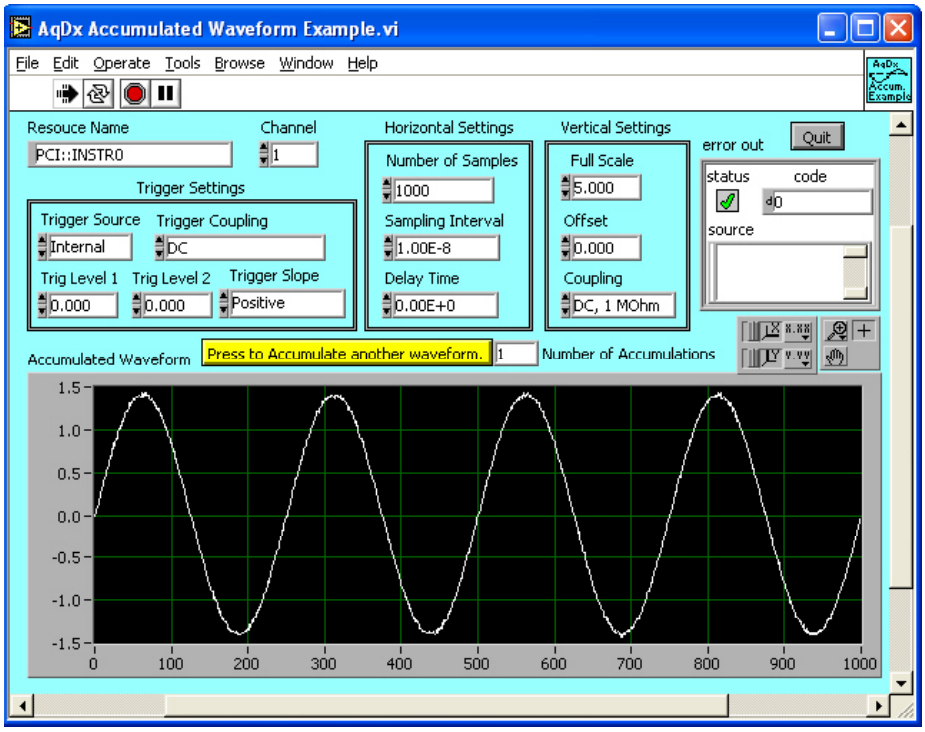
This VI demonstrates how to use some of the basic components of the AqDx Acqiris Digitizer Driver. It finds and initializes a digitizer, sets the basic parameters according to the controls on the front panel, and then acquires one waveform. Note that the front panel controls should be set to their desired values before the VI is run.

2.3.2. Example Scope VI



This VI presents a basic, interactive virtual oscilloscope using the AqDx Digitizer Driver. Not all the functionality of the Acqiris digitizers is supported in this program, but most of the most-commonly-used functions are demonstrated in it.

2.3.3. Accumulated Waveform Example VI



This VI demonstrates how to use the Accumulate Waveform function of the Acqiris Digitizer Driver. It finds and initializes a digitizer, sets the basic parameters according to the controls on the front panel, and then acquires and accumulates waveforms as the user requests. Note that the front panel controls should be set to their desired values before the VI is run.

2.4. Visual Basic

Visual Basic support is available for Versions 5.0 or 6.0 with sample programs in the VB directory. Basic .NET support (AcqrsD1Interface.vb) is also available but there are no sample programs yet.

The Visual Basic sample program comes in 8-bit and 12-bit versions and consists of 3 major parts:

1. The **AcqrsD1Interface.bas** file that contains the Visual Basic version of the interface to the Acqiris supplied driver dll's (installed in the System directory). This file is needed for all Visual Basic projects involving Acqiris digitizers.
2. The **AcqirisShow8.frm** (or **AcqirisShow12.frm**) file that contains all the basic functionality of the sample program, such as initialization, acquisition control and display.
3. The **DevCtrlForm8.frm** (or **DevCtrlForm12.frm**) file with the code for a dialog box for the control of a complete digitizer.

The Visual Basic sample program is capable of managing several Acqiris digitizers attached to the computer and of displaying one channel of one digitizer at a time. If no digitizer exists on the computer, it initializes 3 simulated digitizers (in **Sub Form_Initialize()**). While it has enough functionality to permit a fairly complete operation of a digitizer, many possible features were left out in order to keep the program simple to understand.

In order to run the Visual Basic sample program, you must have installed the Visual Basic Version 5.0 (or 6.0 but not .NET):

- Start Visual Basic
- Use the menu **File → Open Project…** to point to the **AcqirisShow1.vbp** project file in the AcqirisLive\VB directory and choose **Open**
- After the project is loaded, use the menu **Run → Start** or **F5** to start executing the project. A waveform display window should pop up, displaying a real (or simulated) waveform.
- Use the button **Digitizer Control** to make a dialog box appear. The top-most control **Digitizer#** permits the choice of one of several digitizers (and to check how many were found). If no physical

digitizer was found, the program initializes 3 simulated digitizers, a DP110, DC240 and a DC110 (with the 2MB option). You can change the simulated digitizers in **Sub Form_Initialize()**.

- The program is stopped by clicking on the **Close** button (**x**) on the upper right hand side of the waveform display window.

The waveform display window can be resized, but the dialog box cannot.

The acquisition mode **Norm** really is another **Auto** mode, with a slightly longer timeout period. A real **Norm** mode without timeout would require a way for the user to regain control in situations where there is no trigger.

Simulated digitizers have fixed simulated input signals (sine waves, triangle waves or square waves) that cannot be modified through the supplied API.

Note: The programming advice in the rest of this manual is given for the C language interface. However, it is equally valid for Visual Basic. Refer to the file **AcqrsD1Interface.bas** for the correspondence between the Visual Basic and C-language names of the Acqiris driver functions.

2.5. MATLAB

MATLAB is a very powerful environment to analyse and display data. The MathWorks Inc. have developed a Data Acquisition Toolbox that can acquire data from digitizer cards. The Acqiris Adaptor exchanges messages between the MATLAB Engine and the Acqiris instrument driver.

The Acqiris Adaptor lets you acquire data directly from the hardware without additional software. It offers the possibility to start and stop the acquisition as well as to get data. Windows based modules and command line shortcuts are used to configure the instruments.

2.5.1. Examples

These examples are in the directory **<AcqirisDxRoot>\MATLAB**.

When you execute these examples,

either set the MATLAB current directory (**cd** Command) to **<AcqirisDxRoot>\MATLAB**

or copy all of the files from that directory to the directory of your choice.

2.5.1.1. Single Acquisition

Filename: **Aq_SingleAcquisition.m**

Argument: No Argument

This first example shows you how to perform a single acquisition.

The desired setup file is loaded. You then create your AnalogInput object, start the acquisition, and then get and plot the acquired data.

2.5.1.2. Multiple Acquisition

Filename: **Aq_MultipleAcquisition.m**

Argument: No Argument

In this m-file, the main difference with the Single Acquisition example is that we put the acquisition sequence into a loop. In that way, you can manage multiple acquisitions and store data in the MATLAB Workspace.

2.5.1.3. Mode Live

Filename: **Aq_ModeLive.m**

Argument: InstrumentID (Default:0)

Use:
```
>> Aq_ModeLive          // Select default digitizer (ID = 0)
>> Aq_ModeLive(2)       // Select digitizer with InstrumentID = 2
```

This MATLAB program displays your signal in a ‘live’ mode.

An Instrument Settings module that can be used during acquisition has been added. There is also a menu that gives the possibility to load an existing setup file or to save to a new setup file.

You can modify the function applied to the selected channel. You should look for the function named **ProcessFcn** and modify it. You can also add arguments if necessary.

2.5.2. Setup and Settings

2.5.2.1. Manipulating Attributes

In the example called Aq_ModeLive, we offer the possibility to open an Instrument Settings window through which you can modify your instrument attributes. The available attributes are presented in chapter 4 ***ATTRIBUTES***. Please note that not all attributes mentioned there are available for all instruments in all environments. A file with the suffix AqSettings is provided together with the MATLAB examples and is used to define the format of the Instrument settings window.

Note that the Instrument Settings window is not a modal window so it can remain open during acquisitions. Here is the code to display the Instrument Settings window:

```
>> CallAcqiris('InstrumentSettings');
```

Attributes can also be read or modified with commands:

To get all instrument attribute names:

```
>> attributeList = CallAcqiris('AttributeList', ID);
```

To get an attribute's range:

```
>> attrRanges = CallAcqiris('AttributeRanges', ID, 'attrName');
```

To retrieve an attribute value:

```
>> value = CallAcqiris('GetString', ID, 'attrName');
>> value = CallAcqiris('GetInteger', ID, 'attrName');
>> value = CallAcqiris('GetDouble', ID, 'attrName');
```

To modify an attribute:

```
>> CallAcqiris('SetString', ID, 'attrName', 'value');
>> CallAcqiris('SetInteger', ID, 'attrName', value);
>> CallAcqiris('SetDouble', ID, 'attrName', value);
```

2.5.2.2. Loading and Saving a Setup

The way offered by MATLAB to set Properties (Settings) is not very user-friendly, so we decided to offer you the possibility to use our software to configure the settings. Files created with the AcqirisLive program can be used.

To load a setup:

```
>> CallAcqiris('LoadConfiguration'); // Prompt for file
>> CallAcqiris('LoadConfiguration', 'Filename');
```

To save a setup:

```
>> CallAcqiris('SaveConfiguration'); // Prompt for file
>> CallAcqiris('SaveConfiguration', 'Filename');
```

2.5.2.3. Calibration

We added the possibility to do a calibration.

```
>> CallAcqiris('Calibrate', ID, CalibrationType);
```

where ID is the module identifier (0,1,2…)

and CalibrationType can take one of the 3 values:

= 0 calibrate the entire instrument

= 1 calibrate only the current channel configuration

= 2 calibrate external clock timing. Requires operation in External Clock (Continuous).

2.5.3. Process an acquisition

2.5.3.1. Create the AnalogInput object

```
>> AI = analoginput('Aq', 0);
>> BUFFER_SIZE = 1000
>> BUFFER_CNT = 30
```

This command creates an AnalogInput object. The first argument specifies that MATLAB should use the Acqiris Adaptor. The second argument is the InstrumentID of the digitizer to use. The two BUFFER constants are initialized for use in 2.5.3.4 below.

Alternatively,

```
>> AI = analoginput('Aq', 0, 'cal=false');
```

The option added tells the driver to skip the calibration.

A calibration is necessary when first using an AnalogInput object in a MATLAB session. However, since the calibration settings remain in effect until the driver is unloaded (e.g. when closing MATLAB) this can be used to avoid recalibration at each execution of an m-file. For best measurement quality, the calibration should be repeated after any significant temperature drift.

Note that unless calibration is deactivated, all detected instruments will be calibrated.

2.5.3.2. View the Instrument Information

```
>> daqhwinfo(AI);
```

This call will display information about your digitizer.

2.5.3.3. Configure and Add Channels

```
>> AddChannel(AI, i);
```

where i = 1…n. This command is used to create a channel. You can create as many channel objects as there are channels on the given instrument.

2.5.3.4. Start an Acquisition

```
>> set(AI, 'BufferingConfig', [BUFFER_SIZE, BUFFER_CNT]);
>> start(AI);
>> waittilstop(AI, TIME_OUT);
```

⚠ ***NOTE:*** *Although you don't need to understand the first command, be sure that this command is present for each start(AI) because it is needed for engine buffer allocation and it must be done if samples or segments have been changed. Appropriate values should be set once and subsequently reused. The number of buffers must be at least 2*!

The second command starts the acquisition and the third waits for the end of the acquisition. You can specify a timeout. If the timeout occurs, MATLAB terminates the file execution with an error message.

2.5.3.5. Read the Data

Support for Adaptor properties is very limited. The following properties are available from the Adaptor in read mode:

- SampleRate
- SamplesPerTrigger

⚠ ***NOTE: Its value corresponds to samples*segments***

- segments
- segmentDescrIndex,

Using the property segmentDescrIndex, you can access:

- horPos
- timeStampLo
- timeStampHi

Refer to the section 3.10, **Horizontal Parameters in Acquired Waveforms** and section 3.11, **Sequence Timestamps** for detailed explanations on the interpretation of these values.

For averagers we provide:

- nbrAveragedWaveforms (from DataDescriptor)

The simplest way to read data for an acquisition with a single channel in a single segment is:

```
>> [DataArray,Time] = getdata(AI);
>> set(AI.channel(0),'segmentDescrIndex', 0);
>> trDelay = CallAcqiris('GetDouble', ID, 'trDelay');
>> Time = Time + get(AI.channel(0),'horPos') + trDelay;
```

The first command returns all the data acquired together with a vector containing the time values for each sample acquired. The DataArray will contain the measured values converted to the user units as follows:

valueInUserUnit = userGain * voltData + userOffset

voltData = vGain * rawData + vOffset

where

userGain = chScaleFactor * chScale2ndFactor * $10^{chAttenuation/20}$

userOffset = chScaleOffset

vGain = chFullScale/$256^{dataSize}$

dataSize = size in bytes of a rawData sample (1 or 2, depending on the instrument)

vOffset = chOffset

The original Time vector starts at 0 and increases in steps of 1./SampleRate. The last line above shows the horPos and trDelay corrections to the Time vector so that the time of the trigger is 0.

For segmented acquisition, we decided to concatenate all segments in the same vector returned by the method getData(). For multichannel instruments the getData() method returns a vector for each channel activated. So if you have chosen to acquire on n channels, m segments of p samples, you will get a matrix with size: [m*p, n]. Using MATLAB commands you can easily extract only one segment from the matrix. For example, if you would like to extract the segment j on the channel k, you can write:

```
>> vectorSeg = DataAcquired(((j-1)*p +1):j*p, k)
```

The Time vector shown in the simple case above is not appropriate because the adaptor has no information about horPos correction and t0 value. Note that t0 value is different for each channel and horPos is different for each segment. So, if you want to access information for segment j on channel k:

```
>> set(AI.channel(k),' segmentDescrIndex', j)
>> horPos =  get(AI.channel(k),'horPos')
>> timeStampLo =  get(AI.channel(k),'timeStampLo')
>> timeStampHi = get(AI.channel(k),'timeStampHi')
```

Similarly for the t0 value:

```
>> t0 = get(AI.channel(k), 't0')
```

where t0 = trDelay + chTimeOffset +chTime2ndOffset + chTime3rdOffset.

You can use these values to create a ‘real’ time vector where the time of the first data point with resepect to the trigger is given by t0+horPos and the time between samples is as before. The timeStamp values can be combined and used to measure the time difference between any pair of triggers in the acquisition.

2.5.3.6. Free Resources

```
>> delete(AI);
```

It's very important to free allocated resources when they are no longer needed.

2.6. Phar Lap ETS

The sample program is written for the Microsoft Visual Studio environment. Open the project file **GetStartedETS.dsp**. The **GetStartedETS.cpp** file contains a simple user program which:

- finds the Acqiris digitizers on the target machine
- initializes the first (or only) one
- configures some acquisition parameters (and rereads some of them for checking)
- starts the acquisition
- waits for it to terminate
- reads the waveform
- saves a binary disk file on the target with the data

The program can be built, downloaded, and executed/debugged from Visual Studio. If FPGA .bit files are required for the instruments installed they can be transferred to the target's hard disk using the pharemb\monitor utility makehd. The current version appears to work for Acqiris PCI modules only.

2.7. VxWorks (Tornado)

The VxWorks sample program is written for the Tornado environment. The **GetStartedVxW.cpp** file contains a simple user program which:

- spawns a process with a large stack as needed by the Acqiris driver
- finds the Acqiris digitizers on the target machine
- initializes the first (or only) one
- configures some acquisition parameters (and rereads them for checking)
- loops (100 times) over a cycle that
  - starts the acquisition
  - waits for it to terminate
  - reads the waveform

2.7.1. Compiling

This file can be compiled 'as is' within the **GetStartedVxW.wsp** Tornado workspace. However, please make sure to run an update to the project dependencies. You may need to change the directory paths for the Tornado include files and the Acqiris include files if you move the Tornado directory or use a different development environment. In the latter case you must verify that the debug flags

–D_VXWORKS –D_ACQIRIS are present.

2.7.2. Loading

First download the Low-Level (kernel-mode) VxWorks Device Driver file **VxWorksDriverPCI.out** to the target machine, followed by the VxWorks Digitizer Driver file **AcqirisVxWUMode.out**.

Finally, you should download the VxWorks application **GetStartedVxW.out**.

2.7.3. Running the Program

It is recommended that you follow the code with the debugger, since it is the only way to see the response of the function calls immediately.

2.8. Linux

The AcqirisDemo program can simply be run from the AcqirisLinux directory by entering:

Demo/AcqirisDemo

The library /usr/lib/libAnDBSLib.a is compiled with the gcc 3.2.

The Linux **GetStarted** sample program is provided ready to run. It can also be created without the need of an additional development application.

- Enter the command "`cd` **`usr/src`**"
- Run **make** to compile and link the program
- Note that the include and library paths are defined in the **Makefile**.

The **GetStarted.cpp** file contains a simple user program which:

- has globally allocated buffers to achieve optimal readout performance
- finds the Acqiris digitizers on the target machine
- initializes the first (or only) one
- configures some acquisition parameters (and rereads them for checking)
- loops (100 times) over a cycle that
  - starts the acquisition
  - waits for it to terminate
  - reads the waveform

3. Programming an Acqiris Digitizer

3.1. Programming Hints

When programming an Acqiris digitizer it is important to remember that the Acqiris driver must be freshly loaded (this is usually automatic) by any process that uses the modules. This means that each process starts over with a completely clean view of the system and no knowledge of any previously determined calibration constants or settings. Thus a calibration ought to be done before the modules are used for any acquisitions. Of course the system may have to be recalibrated later if the temperature of the modules is changing. Users cannot expect to control Acqiris modules with a succession of process invocations with each one executing a single command. Only one process on a machine can have loaded the Acqiris driver at any given moment.

Most of the AcqrsD1 driver functions are reentrant. After initialization, they are protected against multiple calls from different threads. The unprotected routines are :

AcqirisD1_init	**AcqirisD1_initWithOptions**	**AcqirisD1_getNbrPhysicalInstruments**
AcqirisD1_closeAll	**AcqirisD1_multiInstrAutoDefine**	**AcqirisD1_multiInstrDefine**
	AcqirisD1_multiInstrUndefineAll	

Be sure to read the comments in the header file **AcqirisD1Interface.h** or **AcqrsD1Interface.bas** or the function parameter discussions in chapter 2, **DEVICE DRIVER FUNCTION REFERENCE** of the **Programmer's Reference Manual**.

The examples below do not check the return value of the **AcqrsD1_…** functions. In real applications, you should always check the return values of functions.

3.2. Device Initialization

Before any (real or simulated) device can be used, each device must be initialized with a separate call to the function **AcqrsD1_InitWithOptions**. For real devices, you can also use the slightly simpler function **AcqrsD1_Init**.

Both functions return the **instrumentID** (whose value will be different for each device), which must be subsequently used in any other function call. The arguments **IDQuery** and **resetDevice** are currently ignored. The use of the string arguments **resourceName** and **optionsString** are explained with the initialization scenarios in the following sections.

If you use modules that are connected via ASBus, you need to configure them as *MultiInstruments.* This lets you treat them as normal instruments with an increased number of channels. E.g. you can connect 3 DC270's to form a single 12-channel, 1 GS/s digitizer.

The function **AcqrsD1_multiInstrumentAutoDefine** is recommended for this task. It automatically searches for all sets of modules that are connected with ASBus, and configures each such block as a single *MultiInstrument*. It then reports the total number of instruments found, including individual modules without ASBus connections. You still need to obtain the **instrumentID** for each instrument by calling the function **AcqrsD1_InitWithOptions** afterwards. For details, please refer to the section 3.2.5, **Automatic Definition of MultiInstruments**.

Alternatively, you can initialize each module individually with the function **AcqrsD1_InitWithOptions**, and then combine some of them with the function **AcqrsD1_multiInstrDefine**. This method provides better control over which modules are combined and in what order, at the expense of careful book keeping of which instrumentID's are available. For details, please refer to the section 3.2.6, **Manual Definition of MultiInstruments**.

3.2.1. Identification by Order Found

If you don't know which and/or how many Acqiris digitizers are present on the machine, use this code fragment:

```
ViSession instrumentID[10];
long nbrInstruments;
ViStatus status;
ViString options = "";
status = AcqrsD1_getNbrPhysicalInstruments(&nbrInstruments);
// Initialize the digitizers
for (long i = 0; i < nbrInstruments; i++)
{
     char resourceName[20];
     sprintf(resourceName, "PCI::INSTR%d", i);
     status = AcqrsD1_InitWithOptions(resourceName, VI_FALSE,
                    VI_FALSE, options, &(instrumentID[i]));
}
```

The resource name must be of the form "PCI::INSTR0", "PCI::INSTR1", etc.

If there are several digitizers on the system, the order in which they are found is not obvious. It depends on the Windows 95/98/NT4/2000/XP Configuration Manager implementation, on the PCI bus topology in your computer, and possibly on the BIOS.

3.2.2. Identification by Serial Number

All Acqiris digitizers are labeled with a unique serial number. For PCI digitizers you will find it on the front panel and for CompactPCI digitizers it is on the right injector/ejector handle. This same serial number is coded into an on-board EEPROM that is read by the Device Driver upon initialization. You can therefore ask to initialize a specific digitizer by specifying its serial number:

```
ViSession instrumentID;
AcqrsD1_InitWithOptions("PCI::SER10047", VI_FALSE, VI_FALSE, "",
                    &instrumentID);
```

Note that the serial number must be contiguous to the keyword SER; leading zeros are accepted.

3.2.3. Identification by Bus/Slot Number

While initialization by serial number is easy to implement, it has the drawback that anytime a digitizer is replaced by another one (e.g. if a failure occurred), the program has to be modified. Acqiris offers the possibility of specifying the logical position of the device at initialization:

```
ViSession instrumentID;
AcqrsD1_InitWithOptions("PCI::BUS02::SLOT06", VI_FALSE, VI_FALSE,
                    "", &instrumentID);
```

Again, the bus and slot numbers must be contiguous to the keywords BUS and SLOT; leading zeros are accepted.

Unfortunately, it is not obvious at all by simple inspection, which bus and slot number a given PCI device occupies. One way to find out is to use AcqirisLive and to observe the bus/slot numbers that can be found under the Help menu selection in Instrument Information. Another way is to use the auto-identification initialization method and then to interrogate each device with:

```
ViSession instrumentID;
char name[20];
long serialNbr, busNbr, slotNbr;
AcqrsD1_getInstrumentData(instrumentID, name, &serialNbr, &busNbr,
                          &slotNbr);
```

3.2.4. Firmware initialization (AP-FAMILY/12-bit-FAMILY)

In these modules the on-board FPGA's (field-programmable gate arrays) contain processor logic needed to efficiently execute several crucial functions. For Windows and Linux users, they will be automatically programmed at startup before calibration. The standard initialization using **AcqrsD1_Init** can be used.

The name for the FPGA program file is a synthesis of model, FPGA destination, and option information. The file name suffix is always ".bit". The automatic initialization mentioned above will load the FPGA files as follows:

- First the desired files will be searched for in the working directory of the application.
- Then the working directory will be searched for a file "AqDrv4.ini"
- Finally the directory pointed to by the environment variable "AcqirisDxDir" will be searched for a file "AqDrv4.ini"
- The "AqDrv4.ini" file should contain the name of a directory which will also be searched for the appropriate FPGA files. Here is a typical example of its contents:

```
[Acqiris]
fpgaPath=C:\Program Files\Acqiris\firmware
GeoMapPath=C:\Program Files\Acqiris\bin
```

  The GeoMapPath entry will be described later in this chapter.

Additional Phar Lap ETS/VxWorks Instructions

For ETS and VxWorks users, the normal mechanism for finding the FPGA .bit files will not work; the driver has to be told explicitly where to find them. This procedure is also shown in the **GetStartedETS.cpp** ETS sample program and the **GetStarted.cpp** VxWorks sample program. Thus, **AcqrsD1_InitWithOptions** has to be called with

```
ViString options = "cal=0";
status = AcqrsD1_InitWithOptions(resourceName, VI_FALSE, VI_FALSE,
                          options, &(instrumentID[i]));
```

Then, before using the desired module in any mode, you should execute code like that shown below:

```
ViString FPGADirectoryName = "C:\firmware"; // or "C:\" for ETS
AcqrsD1_configLogicDevice(instrumentID, NULL,
                                FPGADirectoryName, 2);
```

As a final step you should now calibrate the instrument (which will cause the FPGA files to be loaded):

```
Status = AcqrsD1_calibrate(instrumentID);
```

3.2.5. Automatic Definition of MultiInstruments

The function **AcqrsD1_multiInstrumentAutoDefine** automatically searches for all sets of modules that are connected with ASBus, and configures each such block as a single *MultiInstrument*. It then reports the total number of instruments found, including individual modules without ASBus connections. You still need to retrieve the **instrumentID** for each instrument by calling the function **AcqrsD1_InitWithOptions** afterwards, as shown below:

```
ViSession instrumentID[10];
long nbrInstruments;
ViStatus status;
ViString options = "";
status = AcqrsD1_multiInstrAutoDefine(&nbrInstruments);

// Retrieve the digitizer identifiers
for (long i = 0; i < nbrInstruments; i++)
{
     char resourceName[20];
     sprintf(resourceName, "PCI::INSTR%d", i);
     status = AcqrsD1_InitWithOptions(resourceName, VI_FALSE,
                    VI_FALSE, options, &(instrumentID[i]));
}
```

The calls to **AcqrsD1_InitWithOptions** are needed to obtain the **instrumentID's**. The physical digitizers were already initialized when **AcqrsD1_multiInstrumentAutoDefine** was called.

The digitizers within a single *MultiInstrument* are numbered from 0 to (NbrModulesInInstrument – 1). In Acqiris CC10x compactPCI crates, the module 0 is always closest to the controller slot, i.e. module numbers increase with increasing front panel slot numbers. This statement is also applicable to CC121 crate configurations with both an Acqiris acquisition module in one of the last 7 slots and with the PC running under Windows 2000, NT, or XP. Users of other systems or crates may need to provide AqGeo.map files to give the driver needed information. The GeoMapper application described in the Digitizer User Manuals can create this file. For details on channel and trigger source numbering, please refer to section 3.13, **ASBus Operation**.

The master module is automatically chosen, according to these rules:

- If modules of different memory lengths are combined, only modules with the shortest memory length can be master modules
- The master module is chosen as near as possible to the center module, in order to minimize propagation delays.

The function **AcqrsD1_getInstrumentData** will return the information about the master module. If you want control over which module is the master, and in which order they should appear, use the manual definition, described in the next section.

3.2.6. Manual Definition of MultiInstruments

The function **AcqrsD1_multiInstrDefine** permits a 'manual' definition of how to combine multiple digitizers with the ASBus.

Use a code fragment like the following one for the manual combination of digitizers:

```
ViSession instrumentID[10], idList[6];
long nbrInstruments;
ViStatus status;
ViString options = "cal=0"; // since calibration will be performed
                    explicitly later
status = AcqrsD1_getNbrPhysicalInstruments(&nbrInstruments);
```

```
// Initialize the digitizers
for (long i = 0; i < nbrInstruments; i++)
{
     char resourceName[20];
     sprintf(resourceName, "PCI::INSTR%d", i);
     status = AcqrsD1_InitWithOptions(resourceName, VI_FALSE,
                    VI_FALSE, options, &(instrumentID[i]));
}
// Now combine the first 3 digitizers (in inverse order)
idList[0] = instrumentID[2];
idList[1] = instrumentID[1];
idList[2] = instrumentID[0];
ViSession multiInstrID;
ViSession masterIndex = instrumentID[1];
long nbrInList = 3;
status = AcqrsD1_multiInstrDefine(idList, nbrInList, masterIndex,
                    &multiInstrID);
status = AcqrsD1_calibrate(multiInstrID);
```

The first part of the code above finds and initializes all individual digitizers, as shown in section 3.2.1, **Identification by Order Found**. Of course, you could also use one of the other 2 methods of identifying individual digitizers.

After **AcqrsD1_multiInstrDefine** has executed successfully, the **instrumentID's** in the list **idList** become unavailable for further operations. You must use the returned **multiInstrID** to refer to the newly defined *MultiInstrument*.

You are responsible for making sure that

- all participating digitizers are physically connected with ASBus bridges
- the modules are of the same model type
- the master module has no more memory than any other participating digitizer
- an AcqGeo.map file is available if needed by the driver

If the master module has more memory than any other digitizer, the combined instrument will work as long as you never request more memory than that available in the unit with the shortest memory.

The digitizers within the 'manually' defined *MultiInstrument* are numbered from 0 to (nbrInList – 1), exactly as presented in the digitizer list **idList**. For details on channel and trigger source numbering, please refer to section 3.13, **ASBus Operation**.

3.2.7. AqGeo.map file positioning

For ASBus MultiInstruments in some systems the driver will need additional information about the physical ordering of the modules. This information is stored in a file named AqGeo.map which the driver will load when an ASBus instrument is defined. The driver will search for the AqGeo.map file as follows:

- First the file will be searched for in the working directory of the application.
- Then the working directory will be searched for a file "AqDrv4.ini"
- Finally the directory pointed to by the environment variable "AcqirisDxDir" will be searched for a file "AqDrv4.ini"
- The "AqDrv4.ini" file should contain the name of a directory which will also be searched for the AqGeo.map file. Here is a typical example of its contents:

```
[Acqiris]
GeoMapPath=C:\Program Files\Acqiris\bin
```

3.2.8. Simulated Devices

If you want to work with simulated devices, none of the methods above are applicable. Any module supported by the driver can be simulated in its digitizer mode. Any memory option available for the module can be used for a call to **AcqrsD1_setSimulationOptions** as shown in this code fragment:

```
ViSession instrumentID;
ViStatus status;
status = AcqrsD1_setSimulationOptions("M2M");
// Initialize the digitizer
status = AcqrsD1_InitWithOptions("PCI::DC110", VI_FALSE, VI_FALSE,
                    "simulate=TRUE", &(instrumentID));
```

The first function call sets the instrument options that you want to obtain, e.g. "M2M" for the long memory option of a DC110. In the second call, you must specify "simulate=TRUE" (**without** any spaces!). The device driver creates a simulated device of your choice. The resource name string is always of the form "PCI::aannn", where aannn is a valid digitizer module name.

The simulation options will apply to all subsequent calls to **AcqrsD1_InitWithOptions**, until they are reset with **AcqrsD1_setSimulationOptions ("")**.

3.2.9. Terminating an Application

For an orderly shut down of your application, we recommend the following sequence:

```
// Stop the digitizers
for (long i = 0; i < nbrInstruments; i++)
{
     status = AcqrsD1_stopAcquisition(instrumentID[i]);
}
AcqrsD1_closeAll();
```

Stopping the acquisition of all digitizers ensures that there is no further activity that could, for example, generate an interrupt. The function **AcqrsD1_closeAll** shuts down the driver components in the correct order, and thus helps avoid crashes of the application during closing.

3.3. Device Configuration

As a general rule it should be remembered that new values to be used by the modules, as set with the AcqrsD1_config functions, are remembered by the driver software but not immediately acted upon. They will only really be loaded into the instrument's registers at the beginning of an acquisition when AcqrsD1_acquire is called. At that time all necessary changes will be made and, depending on the type of changes, the driver will force itself to wait the appropriate settling time before it starts the acquisition. This is done to ensure that the acquisition will occur in the desired state. The program does not have to include *ad hoc* waits to allow the digitizer hardware to settle. The settling times vary from none in the case of no change, to ~0.5 ms in the case of offset changes and to ~5 ms for relay changes or changes between normal operation and External Clock Reference. In the special case of switching from DC to AC coupling, the settling time is 100 ms. Time base setting changes also have associated settling times.

⚠ ***NOTE:*** *The special case of* ***making transitions from low to high impedance*** *is treated immediately to avoid the risk of damaging the front-end circuitry of the digitizer.* ***When making transitions from high to low impedance you must ensure that large voltages are not applied before the change has really happened****. Therefore it is recommended to wait ~5 ms after having asked for an impedance change and before applying any large voltages. Impedance changes can also affect other devices in the signal path.*

Unneeded calls to the AcqrsD1_config functions should be avoided because they can delay the start of the next acquisition.

⚠ ***NOTE:*** *The AcqrsD1_get functions return the configuration values to be used for the next acquisition.*

Use the following short code fragment for a device configuration:

```
// Configure
double sampInterval = 1.e-9, delayTime = 0.0;
long nbrSamples = 10000, nbrSegments = 1;
long channel = 1, coupling = 1, bandwidth = 0;
double fullScale = 2.0, offset = 0.0;
long trigCoupling = 0, trigSlope = 0;
double trigLevel = 20.0; // in % of vertical Full Scale !
AcqrsD1_configHorizontal(instrumentID, sampInterval, delayTime);
AcqrsD1_configMemory(instrumentID, nbrSamples, nbrSegments);
AcqrsD1_configVertical(instrumentID, channel, fullScale, offset,
                       coupling, bandwidth);
AcqrsD1_configTrigClass(instrumentID, 0, 0x00000001, 0, 0, 0.0,
                       0.0);
AcqrsD1_configTrigSource(instrumentID, channel, trigCoupling,
                       trigSlope, trigLevel, 0.0);
```

Comments:

- Channel numbers run from 1 to nbrChannels, not from 0! Segment numbers, however, run from 0 to (nbrSegments – 1).
- Specifying more than 1 segment in **AcqrsD1_configMemory** implies the use of Sequence mode.
- The 5 main configuration functions are protected against illegal or incoherent values. Thus, the system might adapt the values you ask for. There are 5 'query' counterparts to these functions, **AcqrsD1_getHorizontal, AcqrsD1_getMemory, AcqrsD1_getVertical, AcqrsD1_getTrigClass,** and **AcqrsD1_getTrigSource**, which you can interrogate.
- The function **AcqrsD1_configTrigClass** configures the trigger class control parameters of the digitizer. On current Acqiris products, the edge trigger class is the only class available. For this class, the available source patterns are Channel 1 through 4 or the external trigger. The **AcqrsD1_configTrigSource** function configures the source parameters coupling, slope, and level as shown in the example above. Notice that the functions **AcqrsD1_configTrigClass** and **AcqrsD1_configTrigSource** must always be used together in order to complete the setup of the trigger configuration. Refer to chapter 2, **DEVICE DRIVER FUNCTION REFERENCE** of the **Programmer's Reference Manual** for a detailed description of these two functions.
- The helper functions **AcqrsD1_bestSampInterval** and **AcqrsD1_bestNominalSamples** are sometimes useful for deciding on the nominal number of data points and the sampling interval to use for a given time window to cover. If you ask for a nominal number of samples, the system actually needs some additional samples for reasons of data alignment, acquisition stop-time overhead and other reasons. In some cases, the additional 'invisible' samples can exceed the number of 'visible' ones. The helper functions take such memory overheads into account when advising you on the recommended sampling interval and number of samples. You are free to ignore the advice, but the system is likely to adapt your setup values if the requested number of samples does not fit the available memory.
- Specifying the value 0 for **delayTime** sets the trigger point to the beginning of the waveform. A negative value corresponds to pre-trigger, a positive one to post-trigger. Refer to the section 3.9, **Trigger Delay and Horizontal Waveform Position** for a detailed explanation of the use of **delayTime**.
- For DC coupling the trigger levels in %FS as needed by **AcqrsD1_configTrigSource** can be calculated as follows:

  ```
  TriggerLevelPercent = 100*(TriggerLevelVolts + vOffset)/Fsrange;
  ```

- The granularity of a trigger value setting is limited by the hardware that uses an 8-bit DAC covering somewhat more than the desired range.
- To set the external trigger range for a DC271-FAMILY module or an AP240/AP235 signal analyzer platform, add a call to **AcqrsD1_configVertical** with channel = -1 before the call to **AcqrsD1_configTrigSource**.

3.4. Configuring Averagers

3.4.1. Basic configuration

The averagers have 2 operational modes, *digitizer* and *averager*, controlled with the function:

```
AcqrsD1_configMode(instrumentID, mode, 0, 0);
```

The value *mode* can be set to 0 (digitizer) or 2 (averager).

The averager mode uses a number of additional configuration parameters, which describe the requested averaging conditions.

Use the following short code fragment to configure an average of 1000 waveforms of 20'000 data points, with a dithering range of ± 15 ADC LSB's and a start delay of 128 samples for an AP100:

```
// Configure
long channelNbr = 0, nbrSamples = 20000, nbrWaveForms = 1000;
long ditherRange = 15, trigResync = 1;
long startDelay = 128, stopDelay = 0;

AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "NbrWaveforms", &nbrWaveForms);
AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "DitherRange", &ditherRange);
AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "TrigResync", &trigResync);
AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "StartDelay", &startDelay);
AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "StopDelay", &stopDelay);
```

Comments:

- The value *channelNbr* is usually 0. However, for AP240/AP235 Signal Analyzer platforms it can take on the value of the desired channel for some of the parameters.
- When in averager mode, the following digitizer parameters are ignored:
- *delayTime* of the function **AcqrsD1_configHorizontal** is replaced by "StartDelay" and "StopDelay" (in future software versions, the parameter *sampInterval* of the function **AcqrsD1_configHorizontal** is likely to be replaced by "SampInterval")
- *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments"
- The values *nbrSamples*, *startDelay* and *stopDelay* must be integer multiples of the 'averaging block size', which is ***always*** 16 in the AP100 or AP240/AP235 Dual-channel mode, and 32 in the AP200 or AP240/AP235 Single-channel mode. If the supplied value is not an integer multiple of the 'averaging block size', it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 250 will be truncated to 240 (15 * 16) on an AP100, and to 224 (7 * 32) on an AP200. You can query the actual value with the function **AcqrsD1_getAvgConfig**.
- The value *startDelay* controls the time between the trigger and the first data sample that is to be added to the averager sum. It is expressed in samples and must be an integer multiple of the 'averaging block size', which is ***always*** 16 in the AP100 or AP240/AP235 Dual-channel mode, and 32 in the AP200 or AP240/AP235 Single-channel mode.
- The *stopDelay* permits the addition of an extra delay to the dead time between the averaging of subsequent waveforms. Its minimum value may be zero. It also is expressed in samples and must be an integer multiple of the 'averaging block size'.
- The *ditherRange* value may be between 0 (no dithering) and 15 (max dithering). Please refer to the next section for further explanations.

3.4.2. Dithering

Dithering reduces the effect of non-ideal differential non-linearity of the analog-to-digital converter, by adding or subtracting small offsets to the input signal. The offset is constant during the acquisition of a single waveform, and then modified to another value during the next waveform.

The dithering range, N is programmable between 0 (no dithering) and 15, with the function

```
AcqrsD1_configAvgConfig(instrumentID, channelNbr,
                        "DitherRange", N);
```

Dithering reduces the range of the ADC by N levels at the top and another N levels at the bottom. In order to avoid any undesirable effects, you should make sure that the signal range of interest is within the reduced ADC range.

3.4.3. 'Fixed Pattern' Background Subtraction

If an averaging operation is executed while the input is open or no signal is applied, the averaged waveform should tend to a constant value with a standard deviation $\sigma_{avg} = \sigma / \sqrt{N}$, where N is the number of waveforms in the average, and σ is the standard deviation of a single waveform.

In reality, only random noise sources are averaged out, while those that are coherent with the sampling clock are not reduced. The open-input averaged waveform thus represents the 'fixed pattern' background of the averager. Subtracting this waveform from each subsequently acquired averaged waveform should result in more precise data.

In order to facilitate the acquisition of the 'fixed pattern' background, the Averager modules offer the following possibilities:

- disconnection of the input with the value *coupling* = 0 in the function **AcqrsD1_configVertical**
- acquisition of an averaged waveform ***without*** a trigger signal with the value 'TrigResync' = 2 (free run) in the function **AcqrsD1_configAvgConfig**. However, you get a better measurement of the 'fixed pattern' background if you acquire with the same trigger conditions as the averaged waveforms that will be corrected. Typically, it is better to continue using an external trigger signal, rather than 'TrigResync' = 2.

The 'fixed pattern' background should be acquired in the same conditions as the averaged waveforms that will be corrected. In particular, the dithering range and the number of waveforms should be the same.

Use the following code fragment to acquire a 'fixed pattern' background with a free- running trigger:

```
const long channelNbr = 0;
double fsr, offset;
long coupl, bwidth, reSync, freeRun = 2;

// Make an acquisition, at the current conditions, but with
// "Grnd" coupling and free running trigger.
AcqrsD1_getVertical(instrID, 1, &fsr, &offset, &coupl, &bwidth);
AcqrsD1_configVertical(instrID, 1, fsr, offset, 0, bwidth);

AcqrsD1_getAvgConfig(instrID, channelNbr, "TrigResync", &reSync);
AcqrsD1_configAvgConfig(instrID, channelNbr, "TrigResync",
                        &freeRun);

AcqrsD1_acquire(instrID);
long timeOut = 1000;          // depends on conditions!
AcqrsD1_waitForEndOfAcquisition(instrID, timeOut);
```

```
long nbrPoints = ???;   // Should be the 'current' number of
                        points!
long timeStampLo, timeStampHi, nbrReturnedSamples;
double horPos, sampTime;

// Read the Waveform directly to the Background buffer
double bckGndWform[nbrPoints];
AcqrsD1_readRealWform(instrID, 1, 0, 0, nbrPoints, bckGndWform,
&nbrReturnedSamples, &horPos, &sampTime, &timeStampLo,
                        &timeStampHi);

// Restore the settings of the averager
AcqrsD1_configVertical(instrID, 1, fsr, offset, coupl, bwidth);
AcqrsD1_configAvgConfig(instrID, channelNbr, "TrigResync",
                        &reSync);
```

Use the following code fragment to acquire a 'fixed pattern' background, assuming that the external trigger can be used and is already set:

```
const long channelNbr = 0;
double fsr, offset;
long coupl, bwidth, reSync, freeRun = 2;

AcqrsD1_acquire(instrID);
long timeOut = 1000;         // depends on conditions!
AcqrsD1_waitForEndOfAcquisition(instrID, timeOut);

long nbrPoints = ???;   // Should be the 'current' number of
                        points!
long timeStampLo, timeStampHi, nbrReturnedSamples;
double horPos, sampTime;

// Read the Waveform directly to the Background buffer
double bckGndWform[nbrPoints];
AcqrsD1_readRealWform(instrID, 1, 0, 0, nbrPoints, bckGndWform,
&nbrReturnedSamples, &horPos, &sampTime, &timeStampLo,
                        &timeStampHi);
```

The examples above assume that the background and the averaged waveforms are read in Volts. In this case, the background data points are simply subtracted from the averaged waveform.

However, if you read the background and the averaged waveforms as 32-bit sums, with the function

```
long bckGndWform[nbrPoints];          // Background as 32-bit sum
AcqrsD1_readData(instrumentID, channel, &readParams,
                        waveformArray, &wfDesc, &segDesc);
```

you must correct the average as follows:

```
corrWform[i] = waveformArray[i] - bckGndWform[i] +
                        128*nbrAvgWforms;
```

The last term corrects for the fact that the 32-bit data are unipolar and that for display purposes the corrected waveform should be in the middle of the vertical range if the averaged waveform is the same as the background.

3.4.4. Configuring Noise Suppressed Accumulation (NSA)

As discussed in the User Manual Family of Averagers the module can be configured to only accept data above a fixed threshold and, if desired, to shift the data in that case. Since these two values are expressed in Volts and used as ADC counts they have to be converted before use. The User Manual describes this transformation that depends on whether Data Inversion has been enabled. The NSA threshold functionality must be enabled and a threshold defined. If this has been done the NSA base subtraction can also be enabled and will be activated using the defined base value. The order of the calls to AcqrsD1_configAvgConfig is not important since the final decision is taken when the acquisition is started. Here is an example:

```
const long channelNbr = 0;
double fsr,offset, threshold, base;

threshold = - offset; // place the threshold at the middle of the
                      screen
base = threshold - fsr/10.; // place the base one division below
                      the threshold
// set the base and threshold voltage values
AcqrsD1_configAvgConfig(instrID, channelNbr, "Threshold",
                      &threshold);
AcqrsD1_configAvgConfig(instrID, channelNbr, "NoiseBase", &base);

// enable the NSA functionality
AcqrsD1_configAvgConfig(instrID, channelNbr, "ThresholdEnable",1);
AcqrsD1_configAvgConfig(instrID, channelNbr, "NoiseBaseEnable",1);
```

3.5. Configuring SSR Analyzers

3.5.1. Acquisition Parameters

The AP235/AP240 SSR analyzers have 2 operational modes, *normal* and *AutoSwitch*, controlled with the function:

```
AcqrsD1_configMode(instrumentID, mode, 0, 0);
```

The value *mode* can be set to 0 (normal) or 7 (*AutoSwitch*).

The *AutoSwitch* mode requires a number of additional configuration parameters that describe the requested acquisition and readout conditions.

Use the following short code fragment to configure a buffered acquisition sequence of 800 waveforms of 5000 data points, with a start delay of 128 samples:

```
// Common Configure
long nbrSamples = 5000, nbrSegments = 800;
long startDelay = 128, stopDelay = 0;
AcqrsD1_configMode(instrumentID, 7, 0, 0);
AcqrsD1_configAvgConfig(instrumentID, 0, "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrumentID, 0, "NbrSegments", &nbrSegments);
AcqrsD1_configAvgConfig(instrumentID, 0, "StartDelay", &startDelay);
AcqrsD1_configAvgConfig(instrumentID, 0, "StopDelay", &stopDelay);
```

Comments:

- The value of the third and fourth argumenta to **AcqrsD1_configMode** must always be 0.
- When in *AutoSwitch* mode, the following digitizer parameters are ignored:
  - *delayTime* of the function **AcqrsD1_configHorizontal** is replaced by "StartDelay" and "StopDelay"

- *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments" in the function **AcqrsD1_configAvgConfig**.

- The values *nbrSamples*, *startDelay* and *stopDelay* must be integer multiples of the 'block size', which is ***always*** 16 in the AP240/AP235 Dual-channel mode, and 32 AP240/AP235 Single-channel mode. If the supplied value is not an integer multiple of the 'averaging block size', it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 250 will be truncated to 240 (15 * 16) for a Dual-channel acquisition, and to 224 (7 * 32) for a Single-channel acquisition. You can query the actual value with the function **AcqrsD1_getAvgConfig**.
- The value *startDelay* controls the time between the trigger and when the first digitized data sample is stored. It should also be noted that when *startDelay* is 0, the first few data points, 5 in the case of Dual-channel mode and 10 in the Single-channel mode, will always be 0.
- The *stopDelay* permits the addition of an extra delay to the dead time between the acquisition of subsequent waveforms. Its minimum value may be zero.
- Although not shown here, a call to **AcqrsD1_configControlIO** can be made in order to set a trigger veto time to be respected after the receipt of a *Prepare for Trigger* signal on a Control I/O connector. This feature is for AP101/AP201 analyzers only.
- Also not shown, is a call to the function **AcqrsD1_configAvgConfig** to set a timeout value for the automatic completion of a segment in case the real trigger never arrives. This feature is for AP101/AP201 analyzers only.

3.5.2. Readout configuration

There are three possible ways of reading the data when in the SSR mode: raw data, user gates, and threshold gates. In all three cases the entire acquisition must be read; you cannot ask for fewer segments or points. These setting are also controlled through the **AcqrsD1_configAvgConfig** routine and therefore must be prepared before the acquisition is started. They can be set independently for each channel if desired.

For raw data readout you should set

```
long channel = 1, gate = 0;
AcqrsD1_configAvgConfig(instrumentID, channel, "GateType", &gate);
```

For user gate readout you have to define the groups of data samples that you want to read for each segment. If needed new values for the gates can be defined during the acquisition process. They will become effective after the next call to **AcqrsD1_processData** or **AcqrsD1_acquire**. Here is some sample code:

```
long g_gateLengthSum[3];
long g_lastGate[3];

long channel = 1, gate =1;
AcqrsD1_configAvgConfig(instrumentID, channel, "GateType", &gate);

// you can define up to 4095 gates,
// GatePos and GateLength must both be multiples of 4
AqGateParameters configSetupData[100];
long configObj   = SSR_Default;

long gateSize = 1000;
// this will be the size we want to read
g_gateLengthSum[channel] = 0;
g_lastGate[channel] = 1; // a very simple example
```

```
for(int g=0;g<g_lastGate[channel];g++)
{
 // the first gate starts with the first point
 configSetupData[g].GatePos          = g * gateSize;
 configSetupData[g].GateLength   = gateSize;
 g_gateLengthSum[channel] += configSetupData[g].GateLength + 8;
}
status = AcqrsD1_configSetupArray(InstrumentID[0], channel, configObj,
                          g_lastGate[channel], configSetupData);
```

For threshold gate readout you have to define the threshold value in volts. Data values greater than this will be selected for readout. If desired you can use **AcqrsD1_configAvgConfig** with the "InvertData" parameter to choose data values less than the threshold. In addition you can define the number of data values before and after each selected value that you always want to see. This number is in the range 0 to 16. However, the value will always be rounded up to the next highest multiple of 4. If two consecutive selected values are 32 or more samples apart a new gate block will be generated. Otherwise, the current block will be continued. In all cases the data transferred will always be a multiple of 4 samples and it will start on a sample whose time position is a multiple of 4.

```
long channel = 1, gate =2;
AcqrsD1_configAvgConfig(instrumentID, channel,
                          "GateType",    &gate);
long preSamples = 0, postSamples = 0;
double threshold = 0.0; // in Volts
long thresEnable = 1;

status = AcqrsD1_configAvgConfig(InstrumentID[0], channel,
                          "PreSamples", &preSamples);
status = AcqrsD1_configAvgConfig(InstrumentID[0], channel,
                          "PostSamples",&postSamples);
status = AcqrsD1_configAvgConfig(InstrumentID[0], channel,
                          "Threshold",   &threshold);
```

3.5.3. Time stamps

The 'On-board' 10 MHz reference clock is used to increment a counter. The value of the counter is stored after the trigger of each new segment. The value of the counter can be read by the software as shown here:

```
double SSRtimeStamp;
AcqrsD1_getInstrumentInfo(InstrumentID[0], "SSRTimeStamp",
                           &SSRtimeStamp);
```

Since each channel is controlled by its own FPGA the time stamps for the same segment are not necessarily the same for the two channels. The command above works with the stamp of Channel 2.

⚠ In this release if the function is called before the first acquisition has been started the value returned will be 0!

It is possible to reset the time stamp using a hardware signal on the P1 or P2 connectors. This can be done with a call like:

```
Long TSReset = 1;
status = AcqrsD1_configAvgConfig(InstrumentID[0], channel,
                          "P1Control",&TSReset);
```

3.6. Configuring AP101/AP201 Analyzers

The models AP101/AP201 have 2 operational modes, *normal* and *buffered* (also called *dual-memory*), controlled with the function:

```
AcqrsD1_configMode(instrumentID, mode, 0, flags);
```

The value *mode* can be set to 0 (normal) or 3 (dual-memory). In *mode* = 3, the parameter *flags* sets the memory bank into which to acquire (0 or 1).

The buffered mode uses a number of additional configuration parameters that describe the requested buffered acquisition conditions.

Use the following short code fragment to configure a buffered acquisition sequence of 800 waveforms of 5000 data points, with a start delay of 128 samples, into memory bank 1:

```
// Configure
long nbrSamples = 5000, nbrSegments = 800;
long startDelay = 128, stopDelay = 0;
AcqrsD1_configMode(instrumentID, 3, 0, 1);


AcqrsD1_configAvgConfig(instrumentID, 0, "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrumentID, 0, "NbrSegments", &nbrSegments);
AcqrsD1_configAvgConfig(instrumentID, 0, "StartDelay", &startDelay);
AcqrsD1_configAvgConfig(instrumentID, 0, "StopDelay", &stopDelay);
```

Comments:

- The value of the second argument to **AcqrsD1_configAvgConfig** must always be 0.
- When in *buffered* mode, the following digitizer parameters are ignored:
- *delayTime* of the function **AcqrsD1_configHorizontal** is replaced by "StartDelay" and "StopDelay"
- *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments" in the function **AcqrsD1_configAvgConfig**.
- The values *nbrSamples*, *startDelay* and *stopDelay* must be integer multiples of the 'acquisition block size', which is ***always*** 16 in the AP101, and 32 in the AP201. If the supplied value is not an integer multiple of the 'acquisition block size', it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 250 will be truncated to 240 (15 * 16) on an AP101, and to 224 (7 * 32) on an AP201. You can query the actual value with the function **AcqrsD1_getAvgConfig**.
- The value *startDelay* controls the time between the trigger and when the first digitized data sample is stored. The *stopDelay* permits the addition of an extra delay to the dead time between the acquisition of subsequent waveforms. Its minimum value may be zero.
- Although not shown here, a call to **AcqrsD1_configControlIO** can be made in order to set a trigger veto time to be respected after the receipt of a *Prepare for Trigger* signal on a Control I/O connector.
- Also not shown, is a call to the function **AcqrsD1_configAvgConfig** to set a timeout value for the automatic completion of a segment in case the real trigger never arrives.

3.7. Data Acquisition

Digitizer operation is preceded by configuring the instrument parameters and then starting the acquisition sequence. New settings are only loaded into the module when the acquisition is started; there is one exception to this rule as discussed for analyzer user gate definition in section 3.5.2 **Readout configuration**.

Similarly, you initiate an averaging operation by configuring the instrument parameters, including those that control the averaging, and then starting the combined acquisition/averaging sequence. The Averager module resets the accumulation buffers and then acquires the requested number of waveforms, each preceded by a front-panel trigger signal, without any software intervention. The **AcqrsD1_acquireEx** function allows an AP100/AP200 Averager to acquire additional data without resetting the accumulation.

Until the operation is terminated, your application is free to execute other tasks. There are several methods of detecting when the acquisition/averaging operation has ended. Finally, you read the averaged waveform with the function **AcqrsD1_readData** as described below.

If you want to acquire several (averaged) waveforms under the same conditions, there is no need to call the **AcqrsD1_config…** functions again. It is sufficient to execute a loop over the "start, wait, read" functions. In principle a subsequent start will happen considerably faster than the first one that was required to load the full configuration.

3.7.1. Starting an Acquisition

Use the following line of code for starting an acquisition:

```
AcqrsD1_acquire(instrumentID); // start the acquisition
```

One such command is required for each module in use. However, if several digitizers are combined to a single *MultiInstrument* with ASBus, only a single command is needed for the combined instrument.

3.7.2. Checking if Ready for Trigger

If many modules are being used it may be useful to know when they are all ready to accept a trigger. This can be done by verifying that they are all finished with their pre-trigger phase (PreTrigger = 0) by using the call below to all instruments (or the last instrument started):

```
AcqrsD1_getInstrumentInfo(ID,"IsPreTriggerRunning",&PreTrigger);
```

3.7.3. Waiting for End of Acquisition

Waveforms cannot be read from the digitizer until the acquisition is terminated. The application may wait for an acquisition to end either by *polling* or by *waiting for interrupt*.

(A) Simple Polling: use the following code fragment for polling the interrupt status:

```
int done = 0;
long timeoutCounter = 100000;
while ((!done) && (--timeoutCounter > 0))
    AcqrsD1_acqDone(instrumentID, &done); // poll for status

if (timeoutCounter <= 0) // timeout, stop acquisition
    STOP ACQUISITION
```

NOTE: The code above has the disadvantage of wasting CPU time while checking the instrument status during the entire acquisition period. In addition, the timeout counter value should be set according to the expected acquisition time, but the loop time depends on the CPU speed.

(B) More Efficient Polling: use this code fragment to release the polling thread for short periods:

```
int done = 0;
long timeoutCounter = 100;
while ((!done) && (--timeoutCounter > 0))
{
    AcqrsD1_acqDone(instrumentID, &done); // poll for status
    Sleep(1);
}
if (timeoutCounter <= 0) // timeout, stop acquisition
     STOP ACQUISITION
```

This code puts the polling thread to sleep for periods of 1 ms at a time, letting other threads of the application or other applications use the CPU time. Setting the timeout counter to 100 means that a total timeout period of 100 ms is expected.

NOTE: This method still has some drawbacks:

- depending on the operating system, the 'Sleep' method often has a granularity of 10 ms or more, rounding any smaller number up to this minimum value
- the response time of the application to the end of acquisition is 50% of the sleep time, on average. With a granularity of 10 ms, the mean latency is therefore 5 ms. Thus, no more than 200 waveforms

per second could be acquired, because the application wastes time waiting for the acquisition to terminate.

(C) Waiting for Interrupt:

```
ViStatus status;
long timeOut = 100; // in ms
status = AcqrsD1_waitForEndOfAcquisition(instrumentID, timeOut);
if (status == ACQIRIS_ERROR_ACQ_TIMEOUT) // timeout, stop
  STOP ACQUISITION
```

This method combines low CPU usage with very good response time:

The function enables the digitizer's 'end-of-acquisition' interrupt and sets up a semaphore that waits for this interrupt. It then releases the thread by 'going to sleep', thus letting other threads of the application or other applications use the CPU time. The function returns as soon as the interrupt occurs or when the timeout expires.

Acqiris ***recommends using*** **AcqrsD1_waitForEndOfAcquisition** since it is the most efficient method. The interrupt latency is of the order of several µs, and no CPU time is wasted.

3.7.4. Stopping/Forcing an Acquisition

The previous section shows a case where an ongoing acquisition must be stopped, typically because there is no trigger. Also, in some situations you may want to use the digitizer to generate a system trigger under software control.

If you still would like to have a valid snapshot of the current input signal, you should generate a trigger signal by software, with the function **AcqrsD1_forceTrig** or **AcqrsD1_forceTrigEx**. Typically, the acquisition does not stop immediately, since the digitizer may continue acquiring some additional data, depending on the **delayTime** and the data acquisition time that were initially configured. Thus, the application should again wait for the acquisition to terminate. Forcing a trigger does not make sense for averagers and analyzers and should not be done. **AcqrsD1_forceTrigEx** allows you to generate a trigger out signal which can be synchronized with the sampling clock if desired.

Use the following code fragment to replace `STOP ACQUISITION` in the previous section:

```
{
    AcqrsD1_forceTrig(instrumentID);
    if (AcqrsD1_waitForEndOfAcquisition(instrumentID, timeOut) ==
        ACQIRIS_ERROR_ACQ_TIMEOUT)
    {
        AcqrsD1_stopAcquisition(instrumentID);
        SCREAM, because a major error occurred
    }
}
```

Note that no timeout should ever occur when waiting for a 'forceTrig' to terminate, provided that the **timeOut** value was made large enough. If a timeout does occur, this would indicate a failure in the digitizer or the entire system.

For users generating triggers under software control it may be desirable to do the data readout in a way that just gives the acquired data points and ignores the correction of the data gotten from the **horPos** measurement of the time from the trigger to the next data sample. This can be done using the **flags** parameter of the **AqReadParameters** structure.

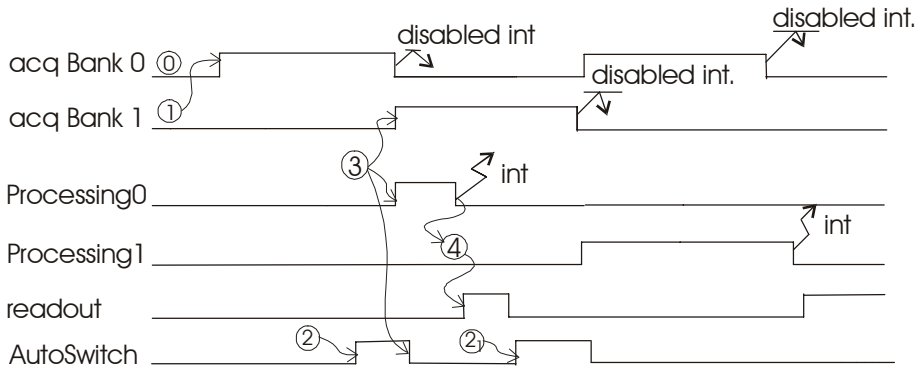
3.7.5. Analyzer Autoswitch mode

The AP Analyzers implement an *autoswitch* mode that allows the dead time between acquisitions to be reduced to the minimum consistent with the readout of the data. As usual the first acquisition must be initialized with a call to **AcqrsD1_acquire(instrumentID**). To allow the second acquisition to start as soon as possible a call to **AcqrsD1_processData (instrumentID,processType,1)** follows immediately. After this start-up procedure data can be read as soon as the processing is terminated and then the go ahead for the next acquisition can be given as desired.

The *AutoSwitch* semaphore is set by the software and cleared by the FPGA.

If the readout process is longer than the acquisition process, the *AutoSwitch* occurs directly after the software raises the *AutoSwitch* semaphore.

At the moment that the ProcessingEnd interrupt occurs, the FPGA has already cleared the *AutoSwitch*.



S	Description	Software implementation
0	The software configures the mode, the acquisition parameters, and the readout.	**AcqrsD1_configMode**(…); **AcqrsD1_configAvgConfig**(…); …
1	The software starts the first acquisition.	**AcqrsD1_acquire**(instID);
2	The software sets the *AutoSwitch* semaphore. To ensure the shortest response time this arming function should be done on the order of 10-20 µs before the expected acquisition end.	**AcqrsD1_processData**(instID, 0, 1);
3	When the acquisition has finish, the FPGA automatically switches the banks, starts a new acquisition, a new processing and clears the *AutoSwitch* semaphore.	
4	Once the software receives the ProcessingEnd interrupt, it can start the readout.	status = **AcqrsD1_waitForEndOfProcessing**(instID, timeout); **AcqrsD1_readData**(instID, channel, &readParams, waveformArray, &wfDesc, &segDesc);
	Go to 2 to continue.	

Here is a sample bit of code showing this principle:

```
status=AcqrsD1_acquire(InstrumentID); // Start the acquisition
processType = 0;

for (;;)  //loop forever
{
     status=AcqrsD1_processData(InstrumentID, processType, 1);
     status=AcqrsD1_waitForEndOfProcessing(InstrumentID,timeout);
     status=AcqrsD1_readData(InstrumentID, channel, &readPar,
                    &adcArray, &dataDesc, &segDesc);
}
```

3.7.5.1. *AutoSwitch* mode without processing

To keep the same interface with and without processing, the processing step is kept but can be reduced to a "dummy" processing. Thus, the software must wait for the end of processing even if the "processing mode" is set to NO_PROCESSING.

3.7.5.2. What happens when the *AutoSwitch* semaphore is not set

After the "processing" of an acquisition, if the semaphore is not set, the FPGA waits for further instructions. This feature ensures that the software has finished with the old buffer and gives full compatibility with older software implementations. If you make a call to **AcqrsD1_stopAcquisition** you shouldn't try to read the last acquisition's data.

3.8. Data Readout

For the reading of standard waveforms the **AcqrsD1_readData** routine should be used. The following older routines will remain available but will no longer be discussed:

AcqrsD1_readCharWform

AcqrsD1_readCharSequence **AcqrsD1_readRealWform**

AcqrsD1_readRealSequence **AcqrsD1_accumulateWform**

You should use the function **AcqrsD1_readData** for all new programs. The older functions will not give support for new instruments or new functionality. All variables of the **AqReadParameters** structure should be initialized; 0 can be used for the reserved words.

For the readout of the averager data the read function **AcqrsD1_readData** described in 3.8.1 **Reading Digitizer Waveforms with the Universal Read Function** should be used with **readMode** = 2. For reading data from analyzers please refer to 3.8.6 **Reading SSR Analyzer Waveforms** or 3.8.7 **Reading AP101/AP201 Analyzer Waveforms**.

3.8.1. Reading Digitizer Waveforms with the Universal Read Function

For the general case, which includes the reading of more complex waveforms, we provide a universal read function **AcqrsD1_readData**.

Control of the read parameters is passed through the input structure **AqReadParameters**. For the description of the output data an array of segment descriptors, **AqSegmentDescriptor,** and a waveform descriptor, **AqDataDescriptor,** are returned. These structures are defined in the header file **AcqirisDataTypes.h**.

The following parameter setting can be used for reading a single waveform segment in 8-bit representation.

```
static long nbrSegments = 1; // readMode = 0 requires this value
long nbrPoints = 1000;
char dataArray[nbrPoints+32];

AqReadParameters     *readPar    = new AqReadParameters;
AqDataDescriptor     *dataDesc   = new AqDataDescriptor;
AqSegmentDescriptor  *segDesc    = new
                     AqSegmentDescriptor[nbrSegments];
readPar->dataType = 0; // 0 = byte
readPar->readMode = 0;  // 0 =  standard waveform
readPar->nbrSegments = nbrSegments;
readPar->firstSampleInSeg = 0;
readPar->segmentOffset = 0;       // unused parameter
readPar->firstSegment = 0;
readPar->nbrSamplesInSeg = nbrPoints;
readPar->dataArraySize = sizeof(dataArray);
readPar->segDescArraySize =
                     sizeof(AqSegmentDescriptor)*nbrSegments;
readPar->flags = 0;
readPar->reserved = 0;
readPar->reserved2 = 0.0;
readPar->reserved3 = 0.0;

status = AcqrsD1_readData(instrumentID, channel, readPar,
                     dataArray , dataDesc, segDesc);
```

Comments:

- The segment numbers run from 0 to **nbrSegments**-1.
- The value of **segDesc->horPos** is the time interval in seconds between the first data point and the nominal time origin of the trigger delay. It is always in the range [-sampTime, 0]. It is useful for a very precise positioning, to a fraction of the sampling interval, of the waveform. In many applications, it can be ignored. Refer to section 3.10, **Horizontal Parameters in Acquired Waveforms**, for a detailed explanation of **horPos**Return values have to be interpreted in the same way as for the other readout functions.
- Refer to the section 3.11, **Sequence Timestamps** for detailed explanations on the interpretation of **segDesc->timeStampLo/Hi**.

3.8.2. Reading Sequences of Waveforms

In certain situations, see ***APPENDIX A: ESTIMATING DATA TRANSFER TIMES***, it can be more efficient (in time) to read Sequence Waveforms with readMode = 1. This mode transfers all of the data from the digitizer to the local memory in a single DMA as opposed to calling **AcqrsD1_readData** many times thus using a transfer per segment. The price to be paid is a higher memory requirement.

For dataType = 0 or 1, the amount of memory needed (in bytes) is

$$\text{arraySize} \geq \textit{segmentOffset} * (\textit{nbrSegments}+1) * (\textit{dataType} + 1)$$

with

$$\text{arraySize} \geq (\textit{nbrSamplesNom} + \textit{currentSegmentPad}) * (\textit{nbrSegments}+1) * (\textit{dataType} + 1)$$

and

segmentOffset≥, *nbrSamplesInSeg*

where

- the *currentSegmentPad* depends on the acquisition configuration and can be determined using the following call,
  `AcqrsD1_getInstrumentInfo(ID,"TbSegmentPad",&currentSegmentPad);`
- the *nbrSamplesNom* is the nominal number of samples to record and may be different than what was asked for! It can be determined using the following call,
  `AcqrsD1_getMemory(ID,&nbrSamplesNom,&nbrSegments);`

You have to make sure that you ask for this information after the acquisition configuration has been established.

Similarly, for dataType = 3 the amount of memory needed (in bytes) is

arraySize ≥ 8 * *segmentOffset* * (*nbrSegments+1)*

and with the other two conditions as above.

The following code can be used for reading a waveform sequence in 8 bit representation.

```
long nbrSegments   = 10;
long nbrPoints = 1000;
char *dataArrayP;
long currentSegmentPad;
long nbrSamplesNom,nbrSegmentsNom;

AqReadParameters     *readPar    = new AqReadParameters;
AqDataDescriptor     *dataDesc   = new AqDataDescriptor;
AqSegmentDescriptor  *segDesc    = new
                     AqSegmentDescriptor[nbrSegments];
readPar->dataType = 0; // 0 = byte
readPar->readMode = 1;  // 1 =  sequence waveform
readPar->nbrSegments = nbrSegments;
readPar->firstSampleInSeg = 0;
readPar->segmentOffset = nbrPoints;
readPar->firstSegment = 0;
readPar->nbrSamplesInSeg = nbrPoints;
readPar->flags = 0;
readPar->reserved = 0;
readPar->reserved2 = 0.0;
readPar->reserved3 = 0.0;

status = AcqrsD1_getInstrumentInfo
                     (ID,"TbSegmentPad",&currentSegmentPad);
// in this case the next call doesn't have any surprises
status = AcqrsD1_getMemory(ID,&nbrSamplesNom,&nbrSegmentsNom);
readPar->dataArraySize =
               (nbrSamplesNom+currentSegmentPad)*(1+nbrSegments);
```

```
// here we show the malloc explicitly
dataArrayP = (char *)malloc(readPar->dataArraySize);
readPar->segDescArraySize =
                    sizeof(AqSegmentDescriptor)*nbrSegments;
status = AcqrsD1_readData(instrumentID, channel, readPar,
                    dataArrayP, dataDesc, segDesc);
```

Comments:

The explicit malloc call will normally not be repeated for every acquisition. Obviously, a larger than needed allocation is perfectly acceptable. Also, any space allocated this way ought to be returned to the heap at some point.

3.8.3. Averaging Waveforms in a Digitizer

The driver includes 4 functions provided to improve performance when averaging waveforms.

The first pair of functions, **AcqrsD1_averagedData** for any digitizer (and the older **AcqrsD1_averagedWform** for 8-bit digitizers only), are meant *only* for single channel, single segment operation. They average a predefined number of waveforms, taking care of the acquisition loop internally. The client must supply a working array (**dataArray** or **waveformArray**, for internal use) and an accumulation array (**sumArray**). The accumulation array is reset automatically inside the function at the beginning of each call. When the function returns successfully, the accumulation array contains the sample-by-sample sum of the waveforms. To get the average values, the array elements must be divided by the number of acquisitions **nbrAcq**. If, for each acquisition, the trigger does not arrive within the requested timeout after the beginning of the acquisition, the function returns with an error code.

The second pair of functions, **AcqrsD1_accumulateData** for any digitizer (and the older **AcqrsD1_accumulateWform** for 8-bit digitizers only) can be used for multi-channel operation and can be called for each acquisition the user wants to accumulate. It reads the waveform in the module, and performs a sample-by-sample accumulation in the client array. Here again, the client must supply a working array (**dataArray** or **waveformArray**, for internal use) and an accumulation array (**sumArray**). The client controls the acquisition, and must reset the accumulation array appropriately.

In both cases, the allocation of the memory for the working array (**dataArray** or **waveformArray**) has been left to the client for performance reasons. Its size must be at least the requested number of samples **nbrSamples** + 32, for reasons of data alignment. The content of this working array is not meant to be used by the client.

Please note that in both cases, sub-sample timing information (i.e. horPos, see section 3.10, **Horizontal Parameters in Acquired Waveforms**) is not taken into account.

3.8.4. Reading an Averaged Waveform from an Averager

Averaged waveforms can be read out either in Volts, or as 32-bit accumulated sums. In either case, Acqiris recommends to use the general-purpose read function **AcqrsD1_readData**, rather than the 'legacy' function **AcqrsD1_readRealWform**.

3.8.4.1. Averaged Waveforms in Volts

You should use the general-purpose function **AcqrsD1_readData** (or the 'legacy' function **AcqrsD1_readRealWform**). As long as the *mode* is still set to *averager*, either function automatically divides the accumulated waveform sum by the number of acquired waveforms, and returns the result in Volts. They also return zero into the variables *horPos*, *tStampLo* and *tStampHi*, since they are irrelevant in the context of an averaged waveform.

Use this code fragment for the general-purpose function:

```
AqReadParameters      readParams; // Read Definitions
AqDataDescriptor        wfDesc;     // Returned (common) waveform
                                    values
AqSegmentDescriptorAvg  segDesc;      // Returned segment values

long channel = 1, nbrSamples = 20000;
double waveformArray[20000];
readParams.dataType = ReadReal64;           // Request Volts
readParams.readMode = ReadModeAvgW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;                // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.dataArraySize = sizeof(waveformArray);
readParams.segDescArraySize = sizeof(AqSegmentDescriptorAvg);
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;

AcqrsD1_readData(instrumentID, channel, &readParams,
                      waveformArray, &wfDesc, &segDesc);
```

Note: If you call a readout function while the acquisition *mode* is set to *digitizer*, it will return the last acquired single waveform, possibly with some unpredictable results.

Note: The 'raw' sums can be read directly with a different function call (see next section). The relationship between Volts and the raw sum is expressed by the following formula:

```
sum[i] = (volts[i] + offset + FS/2.0) * 256 * nbrWforms / FS
```

with the following definitions:

sum[i]	32-bit integer sum at position i, unipolar (i.e. 0 or positive)
volts[i]	floating point voltage at position i, as returned by the code fragments above
offset	offset in Volts, as set with AcqrsD1_configVertical
FS	full scale range in Volts, as set with AcqrsD1_configVertical
nbrWforms	number of summed waveforms

The value of 'nbrWforms' must be known, i.e. if the averaging process was interrupted before reaching the requested number of waveforms, the formula above yields wrong results. As a check that the correct value of 'nbrWforms' was used, the value of 'sum[i]', before conversion to an integer, must already be very close to an integer.

Use this code fragment for the 'legacy' function:

```
long channel = 1, segmentNumber = 0, nbrSamples = 20000;
long returnedSamples, tStampLo, tStampHi;
double waveformArray[20000], horPos, sampTime;
AcqrsD1_readRealWform(instrumentID, channel, segmentNumber, 0,
                      nbrSamples, waveformArray,&returnedSamples,
                      &horPos, &sampTime, &tStampLo, &tStampHi);
```

3.8.4.2. Averaged Waveforms as 32-bit Sums

You must use the general-purpose function **AcqrsD1_readData**.

Use this code fragment:

```
AqReadParameters   readParams;// Read Definitions
AqDataDescriptor   wfDesc;   // Returned (common) waveform values
AqSegmentDescriptorAvg  segDesc;       // Returned segment values

long channel = 1, nbrSamples = 20000;
long waveformArray[20000];
readParams.dataType = ReadInt32;           // Request 32-bit sums
readParams.readMode = ReadModeAvgW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;               // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.dataArraySize = sizeof(waveformArray);
readParams.segDescArraySize = sizeof(segDesc);
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;

AcqrsD1_readData(instrumentID, channel, &readParams,
                     waveformArray, &wfDesc, &segDesc);
```

The returned data values in *waveformArray* are unipolar, i.e. the raw ADC values are coded as values between 0 and 255, so that the summed data values may run between 0 and 255*N (N= number of waveforms in the sum).

3.8.5. Reading a RT Add/Subtract Averaged Waveform from an Averager

This case is significantly different than the normal averager case described above.

The 'raw' sums now have to be considered as signed values. The relationship between Volts and the raw sum is expressed by the following formula:

```
sum[i] = volts[i] * 256 * nbrWforms / FS
```

with the same definitions as before. However, the user has to understand if the final result corresponds to the desired signal or just half of it.

3.8.6. Reading SSR Analyzer Waveforms

3.8.6.1. SSR Mode Readout Data Format

In all cases data values are returned in the range [–128, +127]. The relationship between Volts and the raw data is expressed by the following formula:

```
data[i] = (volts[i] + offset) * 256 / FS
```

with the following definitions:

data[i]	8-bit ADC value at position i, unipolar (i.e. 0 or positive)
volts[i]	floating point voltage at position i, as returned by the code fragments above
offset	offset in Volts, as set with AcqrsD1_configVertical
FS	full scale range in Volts, as set with AcqrsD1_configVertical

In all cases you must readout the entire acquisition. You cannot ask for a reduced number of segments.

3.8.6.2. Raw data

The complete data can be read out using readMode = ReadModeStdW (0). This mode can be used to read the entire acquisition, which may have many segments. You cannot ask for a partial range of the data. Segment timestamps are not available for raw data readout.

3.8.6.3. Gated data

Data can be read for both user and threshold gate operation using readMode = ReadModeSSRW (7).

The waveform descriptor structure contains the value actualDataSize giving the total number of data bytes read. A time stamp block, measuring the trigger time, will mark the beginning of each segment. Segment timestamps are mixed in with the data and not available through the usual segDesc array. The entire time stamp is a 56-bit integer counting in units of 100 ns.

Here is the Time Stamp format:

Marker block

31..24 (8 bits)	23..0 (24 bits)
Flag = 00000100 (0x04)	Time Stamp MSB
TimeStamp LSB	

The time stamp may be followed by a variable number of blocks of data with the following format:

Gate block

31..24 (8 bits)	23..0 (24 bits)		
Flag = 00000000	Gate position from the origin of the acquisition (not the segment!)		
31..0			
Gate length (number of Data bytes, always a multiple of 4)			
31..24	23..16	15..8	7..0
Data3	Data2	Data1	Data0
…	…	…	Data4

When reading such data you should carefully check that you terminate correctly and do not read beyond the end of the transmitted data nor generate unphysical time coordinates for the data.

3.8.6.4. Waveform storage requirements

When using the routine AcqrsD1_readData you must allocate waveform storage and inform the driver about the number of bytes available.

Raw data readout requires exactly the number of bytes corresponding to the number of segments times the number of data points per segment.

User gate readout for each segment requires 8 bytes for the time stamp and an overhead of 8 bytes for each gate. This must be added to the total number of samples in all of the gates to get the required length for each segment and multiplied by the number of segments to get the waveform array length.

For threshold gate readout the program should allocate the space needed for the worst case. This means 8 bytes for the time stamp, 8 bytes for a gate block header and space for the total number of samples/segment. This must be multiplied by the number of segments to get the waveform array length. It should be noted that the FPGA will generate a single gate block if there are less than 32 data points "below" threshold between two desired data points.

3.8.7. Reading AP101/AP201 Analyzer Waveforms

3.8.7.1. Reading a Buffered Waveform Sequence

This section concerns AP101/AP201 Analyzers ONLY. In *normal* mode, you may read the acquired waveform(s) in the same way as with any other digitizer, as described in the section 5.14, **Data Readout,** in the Standard Manual.

In *buffered* mode, you must use the functions **AcqrsD1_readData** to read out the accumulated waveform sequence, *as a single data record*. E.g. if you configured nbrSamples = 5000 and nbrSegments = 800, you should specify segmentNumber = 0 and nbrSamples = 4'000'000.

Before reading the buffered data, you ***must*** switch to the other memory bank. Typically, you also would start a new acquisition before readout, but it is not required. This is done automatically in the *autoswitch* mode as a consequence of the call to **AcqrsD1_processData** with a non-zero flag value. It can also be done with a call to AcqrsD1_configMode

The read-function returns zero into the dataDesc variables *horPos*, *tStampLo* and *tStampHi*, since they are unavailable in the context of a buffered waveform sequence.

With the function **AcqrsD1_readData**, use this code fragment:

```
AqReadParameters     readParams; // Read Definitions
AqDataDescriptor     dataDesc;   // Returned waveform values
AqSegmentDescriptor  segDesc;         // Returned segment values

long channel = 1, nbrSamples = 4000000;
char waveformArray[4000000];
readParams.dataType = ReadInt8;
readParams.readMode = ReadModeStdW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;                // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;

memoryBank = (memoryBank+1)&0x1;       // switch to other bank
AcqrsD1_configMode(instrumentID, 3, 0, memoryBank);
AcqrsD1_acquire(instrumentID);         //essential!!

AcqrsD1_readData(instrumentID, channel, &readParams,
                     waveformArray, &dataDesc, &segDesc);
```

The returned data array contains the acquired waveforms as a contiguous array. E.g. if you configured nbrSamples = 5000, the data points 'waveformArray[0…4999]' correspond to the first waveform, the data points 'waveformArray[5000…9999]' correspond to the second waveform etc.

3.8.7.2. Reading Gated Waveforms

For reading gated waveforms the actual desired gates should be set with the setup function **AcqrsD1_configSetupArray**. This function should be called before **AcqrsD1_acquire** is invoked to acquire any data that needs to be read using these gates. To read back the gate values, **AcqrsD1_getSetupArray** has to be used. An example for the two routines is shown in the following code:

```
const int NbrGates = 64;
long channelNbr = 0;
long configObj = AvgGate;
long lastGate = NbrGates;
AqGateParameters    gatePara[NbrGates];
for(int i=0;i<NbrGates;i++)
{
     gatePara[i].GateLength = 256;
     gatePara[i].GatePos = i*gatePara[i].GateLength;
}
AcqrsD1_configSetupArray(instrumentID, channelNbr, configObj,
                         gatePara, lastGate);
```

The condition *GateLength* >= 4 is required. Both *GateLength* and *GatePos* must be multiples of 4.

You can read the gate parameters back, with this code:

```
const int NbrGates = 64;
long channelNbr = 0;
long configObj = AvgGate;
long lastGate;
AqGateParameters    gatePara[NbrGates];

AcqrsD1_getSetupArray (instrumentID, channelNbr, configObj,
gatePara, &lastGate);
```

Make sure to use a pointer to the last argument, since it returns the number of gates. *lastGate* cannot exceed the number of gates being written.

To read the gated waveforms, use the function **AcqrsD1_readData**.

```
AqReadParameters     readParams; // Read Definitions
AqDataDescriptor     dataDesc;   // Returned waveform values
AqSegmentDescriptor  segDesc;         // Returned segment values

long channel = 1, nbrSamples = 20000;
char waveformArray[20000];
readParams.dataType = ReadInt8;
readParams.readMode = ReadModeGateW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;                // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;

AcqrsD1_readData(instrumentID, channel, &readParams,
                         waveformArray, &dataDesc, &segDesc);
```

The returned data array contains the acquired waveforms as a contiguous array of **dataDesc–>returnedSamples** bytes.

Note: Make sure that the *waveformArray* is large enough to hold the sum of all *GateLength's* times the *nbrSegments*. As a rule, the *waveformArray* has to have as a minimum size the sum of all the gate sizes times *nbrSegments* (*waveformArray* > $(\sum GateLength) * nbrSegments$) .

Note: If, for each gate, the sum *(GatePos + GateLength)* exceeds the *nbrSamplesInSeg*, *GatePos* is reduced to satisfy (*GatePos + GateLength)* <= *nbrSamplesInSeg*. If this is not sufficient, *GateLength* is shortened to satisfy that condition.

3.8.7.3. Data Processing before Readout

In *buffered* mode. the AP101 offers the capability of processing the acquired data before readout. This operation must be explicitly requested by the application, after the data acquisition has terminated.

Depending on the processing algorithms used, you may have to prepare the data processing by setting the appropriate parameters with the function **AcqrsD1_configSetupArray**.

In the 'peak-detect' mode, the AP101 will return for each gate exactly 2 peaks, first the positive and then the negative one (some of which might be marked as 'invalid' if no valid peak exists!). Thus, you should define the gates in the same way as described in the previous section.

A typical acquisition/processing/readout sequence in autoswitch *buffered* mode would be:

1. Configure the APXXX for appropriate channel, timebase, trigger, and gate parameters.
2. Start the first acquisition.
3. Give the order to switch banks and start the next acquisition and data processing on the current acquisition as soon as possible. The processing can overlap with the data acquisition since it automatically deals with the memory bank that is not selected for acquisition. If you need to read the original data choose the "no processing" option.
4. Wait for the processing to be terminated, read the processed result. Note that the processing will not destroy the originally acquired data.
5. You may now do any additional processing in your computer. However, you cannot read the original data at this point without perturbing the acquisition.
6. Now that you have finished all work with the current data you can loop to (3) above.

A typical acquisition/processing/readout sequence in explicit *buffered* mode would be:

7. Configure the APX01 for appropriate channel, timebase, trigger, and gate parameters.
8. Start the first acquisition.
9. Wait for the first acquisition to terminate.
10. Switch the memory bank and start a new acquisition in the second bank. Note that you must start the new acquisition to make the memory bank switch happen.
11. Start data processing. This can overlap with the data acquisition since it automatically deals with the memory bank that is not selected for acquisition.
12. After processing has terminated, read the processed result. Note that the processing will not destroy the originally acquired data.
13. You may now do any additional processing in your computer. However, you cannot read the original data at this point without perturbing the acquisition.
14. Wait for the new acquisition to terminate.
15. If you need to read the original data, e.g. for diagnostics on the processing algorithm, you may now do so.
16. Loop to (4.) above.

You need to implement a method to interrupt the infinite loop whenever required. However, you should make sure that you leave the acquistion in a well-determined state for future operation.

The explicit acquisition/processing/readout sequence described above is shown in the following code:

```
AqReadParameters        readParams;    // Read Definitions
AqDataDescriptor      dataDesc;        // Returned waveform values
long channel = 1, segmentNumber = 0;
long nbrPeaks = 2 * nbrGates;     // nbrGates is defined by user
long waveformArray[2 * nbrPeaks];
long memoryBank = 0, timeout = 5000;  // timeout = 5 seconds
// Insert whatever is required for Vertical
//      and Trigger configuration

AcqrsD1_configMode(instrumentID, 3, 0, memoryBank);
AcqrsD1_acquire(instrumentID);        // Acquire into bank 0
AcqrsD1_waitForEndOfAcquisition(instrumentID, timeout);
// At this point, you should check the return value!

bool finished = false;
while(!finished)
{
     memoryBank = (memoryBank + 1)&0x1;// switch to other bank
     AcqrsD1_configMode(instrumentID, 3, 0, memoryBank);
     AcqrsD1_acquire(instrumentID);    // start new acquisition
     AcqrsD1_processData(instrumentID, 0, 0);// start processing
     AcqrsD1_waitForEndOfProcessing(instrumentID, timeout);
     // At this point, you should check the return value!

     readParams.dataType = ReadReal64;
     readParams.readMode = ReadModePeak;
     readParams.nbrSegments = 1;
     readParams.firstSampleInSeg = 0;
     readParams.segmentOffset = 0;
     readParams.firstSegment = 0;           // Read first segment
     readParams.nbrSamplesInSeg = 2* nbrPeaks; // pos and neg peak
     readParams.dataArraySize = 2 * sizeof(double) * nbrPeaks;
     readParams.segDescArraySize = 0;
     readParams.flags = 0;
     readParams.reserved = 0;
     readParams.reserved2 = 0.0;
     readParams.reserved3 = 0.0;

     AcqrsD1_readData(instrumentID, channel, &readParams,
                      waveformArray, &dataDesc, NULL);
     //...analyse and store data

     AcqrsD1_waitForEndOfAcquisition(instrumentID, timeout);
     //...read original data if desired
     //...check on loop termination conditions and set 'finished'
}
```

The returned **waveformArray** contains exactly 2**nbrGates* peaks, each of which is described by 2 double precision floating-point values. The first pair of doubles contains the positive peak position, within the gate (in units of samples), and the amplitude (in codes). The second pair of doubles contains the negative peak position and its amplitude. The peak amplitude is a signed number in the range [–128.0,+127.0] with the two extreme values indicating underflow/overflow conditions. The peak

position is normally positive. Negative values are used for warnings. In particular, the value –DBL_MAX indicates that no peak was found. The #define of DBL_MAX can be found in the float.h include file. Each peak contains the following two words:

0 – 63
Peak Position (in samples from start of gate)

0 – 63
Peak Amplitude (in ADC units)

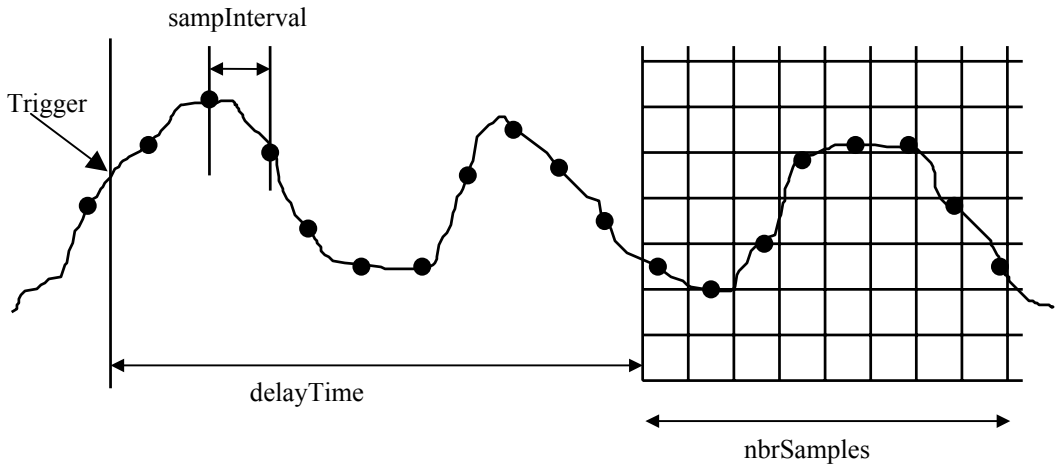
Use the following code to obtain the peak positions for the i'th gate:

```
double *positivePeakPos = &waveformArray[0+8*i];
double *negativePeakPos = &waveformArray[4+8*i];
```

3.9. Trigger Delay and Horizontal Waveform Position

The user has 3 instrument setup variables with which to position the acquired waveform in time:

- **sampInterval**: the sampling interval (inverse of the sampling frequency)
- **nbrSamples**: the number of samples to acquire
- **delayTime**: the nominal trigger delay



By convention, the nominal trigger delay is taken relative to the beginning of the trace, i.e. relative to the left edge of a real or virtual display grid. It can be interpreted as the time from the trigger to the start of waveform recording. If this number is positive, recording starts *after* the trigger (post-trigger acquisition). If it is negative, recording starts *before* the trigger (pre-trigger acquisition). In reality, the acquisition always runs before any trigger occurs, and **delayTime** controls the time between the trigger and the stopping of the acquisition:

delayTime	**Time until Acquisition Stop**	**Comments**
- sampInterval * nbrSamples	0	Trigger point is at the right edge of grid, i.e. at the end of the nominal waveform (100 % pre-trigger)
< 0	**sampInterval * nbrSamples + delayTime**	Trigger point is at the desired point within the grid
0	**sampInterval * nbrSamples**	Trigger point is at the left edge of grid, i.e. at the beginning of the nominal waveform (0 % pre-trigger)
> 0	**sampInterval * nbrSamples + delayTime**	Trigger point is to the left of the grid, i.e. before the beginning of the nominal waveform (post-trigger)

Note that **delayTime** is not allowed to become more negative than - **sampInterval * nbrSamples**, because it is impossible to stop the acquisition *before* the trigger occurs.

3.10. Horizontal Parameters in Acquired Waveforms

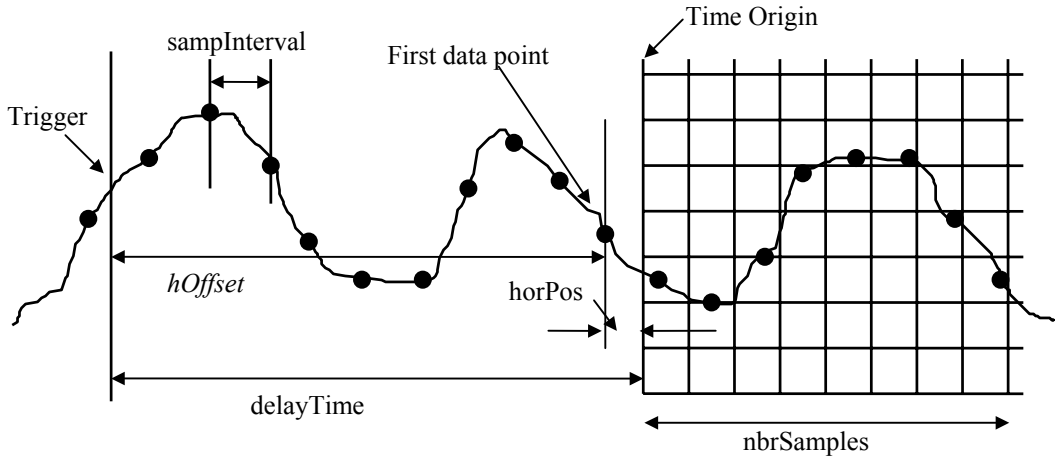
Triggers usually occur asynchronously with respect to the sampling clock. Thus, between similar events, the time from the trigger to the next sampling clock varies randomly in the range [0 … sampInterval].

The true time reference for any waveform is the trigger point, *not* the sampling times, because the trigger is attached to a given feature of the waveform (e.g. a transition at a predetermined level). For highly stable displays, it is important to know the time between the trigger and the next sampling clock to within

a fraction of the sampling interval, and to place the displayed data points in such a way that the trigger point stays at a constant position. This is particularly important for persistence displays or highly zoomed random-interleaved displays, as generated from overlaid segments, where a single waveform (or waveform segment) contributes only a few data points to the display.

Acqiris digitizers feature a Trigger Time Interpolator (TTI), which measures the time between the trigger event and the next sampling clock to a fraction of the sampling interval. It permits very precise positioning of the acquired trace in highly zoomed displays, particularly when multiple acquisitions of the same signal are used. In many other applications, this value can be ignored.

The following drawing completes the description of a 'real-life' waveform:



- The value of **delayTime** positions exactly the left edge of the display (or the exact nominal beginning of the waveform) with respect to the "stable" trigger time, which is the real reference point. We define the time 'trigger time + delayTime' as the time origin for the waveform, which is equivalent to saying that the trigger always occurs exactly at the time **-delayTime**.
- The first data point of the waveform is defined as the *last* acquired data point *before* the time origin. It is indexed with i = 0 in the formula below.
- The exact position of the first data point with respect to the time origin is a negative number **horPos**. It is by definition in the range [**-sampInterval**, 0].
- The time between the trigger and the first data point **hOffset** need not be recorded since it can always be computed as **delayTime + horPos**. Note: **delayTime + horPos** < **delayTime** by definition.
- In order to obtain a very stable image, even in a highly zoomed display, the user only needs to position the acquired data points with the aid of **horPos**, by using the following formula for the x-position of point i with respect to the left edge of the display:

$$x[i] = horPos + i * sampInterval$$

3.11. Sequence Timestamps

In Sequence acquisition mode, multiple waveforms are acquired autonomously, with a single start command **AcqrsD1_acquire**. Whenever a trigger is received, the current acquisition segment is normally terminated. The digitizer then automatically initializes another acquisition into the next memory segment, until all requested segments are filled.

Acqiris digitizers feature a 'timestamp' in order to measure the time between the triggers of consecutive segments. In fact, the timestamp counter is started when the Sequence acquisition is started, and keeps counting during the entire sequence. The difference between the timestamps of any pair of (not necessarily adjacent) segments is the time between their respective triggers.

The timestamp value is returned as a 64-bit integer, in units of picoseconds, with a resolution identical to that of the trigger time interpolator (see the appropriate product User manual). The waveform readout functions **AcqrsD1_readCharWform** and **AcqrsD1_readRealWform** return the timestamp value as 2 32-bit values. In order to do time differences, you should transform them into a 64-bit integer:

- In Visual C/C++, use the 64-integer **__int64** as follows:

```
__int64 timeStamp = timeStampHi;
timeStamp = timeStamp<<32 + (unsigned long)timeStampLo;
```

Arithmetic operations between such integers can be done as with shorter integers.

You also can convert a timestamp difference to an extended floating point number, and do arithmetic operations as with other variables:

```
double deltaTime = (double)(timeStamp - previousStamp);
```

- In Visual Basic, use a decimal **Variant** variable as follows:

```
Const Two16 As Variant = 65536
Const Two32 As Variant = Two16 * Two16
Dim timeStamp As Variant, previousStamp as Variant
Dim timeDiff as Variant, xStampLo as Variant
...
If (tStampLo < 0) Then
    xStampLo = Two32 - Abs(tStampLo)
Else
    xStampLo = tStampLo
End If
timeStamp = CDec(tStampHi * Two32) + xStampLo
...
timeDiff = timeStamp - previousStamp
```

Arithmetic operations between such decimal variants can be done as with other integer variables. The manipulation of **tStampLo** is somewhat complicated because this variable is a signed 32-bit integer, but must be added as an unsigned integer to the (shifted) **tStampHi.**

- In LabVIEW, convert the timestamp to an extended floating point number, and do arithmetic operations as with other variables.
- In LabWindows/CVI, the easiest way to manipulate timestamps is to convert them first to doubles:

```
ViReal64 dlow, dhigh, tstamp;
dlow = (ViReal64)low;
dhigh = (ViReal64)high;
tsamp = dlow + 4294967296.0 * dhigh;
```

3.12. External Clock and Reference

The *external reference mode* replaces the internal 10 MHz reference clock with an external one at the same or a similar frequency, from which the actual sampling clock is derived.

In the *external clock mode*, a waveform is sampled either on transitions of the external clock signal through the user-defined threshold. We distinguish between *continuous* external clock operation and *start/stop* external clock operation.

All external clock/reference modes are configured with the function **AcqrsD1_configExtClock**.

The external clock/reference signal should have an amplitude of at least 2 V peak to peak (1 V for the DC-271 Family, the 12-bit Family, and the AP Averagers and Analyzers). The **inputThreshold** value should be set to the center of the signal.

3.12.1. External Reference

This *external reference* mode (**clockType** = 2) simply replaces the internal 10 MHz reference clock with an external one at the same or a similar frequency. Alternatively, for the DC135/DC140/DC211/DC211A/DC241/DC241A/DC271/DC271A, the PXI 10 MHz System Clock can be used as the reference.

If you need a more precise timebase, or want to ensure that the timebases of several modules are at exactly the same frequency, you should use **clockType** = 2 in the function, and apply an external 10 MHz signal. All other settings of the digitizer are exactly the same as with an internal reference clock.

If you need to sample at a rate that deviates from the nominal values, you may apply an external reference signal with a constant frequency in the range of [9.0, 10.2] MHz. You need to correct for the reference frequency difference in your application since the digitizer and the driver do not take the deviations into account.

⚠ ***NOTE:*** *A square wave with better than 5 ns risetime should be used. This is needed to avoid false or multiple transitions on a slower risetime signal. Alternatively, a >2 V amplitude signal could be used.*

3.12.2. External Clock (Continuous)

The *continuous external clock* mode (**clockType** = 1) permits the application to the digitizer of a continuous, constant frequency, external clock in order to sample at an arbitrary frequency. This mode uses normal triggering, from the input signal or through the external trigger input

We need to distinguish between *first generation* digitizers (models DP105, DP106, DP110, DP111, DP210, DP211, DP212, DC110, DC240, DC265, DC270) and *second generation* digitizers of the DC271-FAMILY (models DC135, DC140, DC211, DC211A, DC241, DC241A, DC271, DC271A, DP214, DP235, DP240), the AP240/AP235 signal analyzer platforms, and the 12-bit-FAMILY (DC440, DC438, DC436, DP310, DP308, DP306) since their behavior in this mode is quite different. *AP200/AP201/AP100/AP101 Averagers and Analyzers* are considered to be *first generation* modules.

The horizontal control parameters **sampInterval** and **delayTime** as defined by **AcqrsD1_configHorizontal** are ignored. You need to give the driver the current input frequency and the requested sampling frequency with the variables **inputFrequency** and **sampFrequency** of the function **AcqrsD1_configExtClock**.

The input frequency **inputFrequency** must be between 10 MHz and 500 MHz in the first generation models, while it must be between 20 MHz and 2000 MHz for the second generation DC271-FAMILY or AP240/AP235 signal analysis platforms or, for the 12-bit FAMILY, between 100 MHz and the value for the maximum allowed sampling frequency. Note that for normal operation of the 12-bit-FAMILY digitizers the sFmax for each converter should be kept above 50 MHz.

The acceptable values for **sampFrequency** are dividers (sFmax/n) of the maximum allowed sampling frequency, sFmax, where n = 1,2,4,8,20,40,80,200,… The sFmax depends on the model and on the number of combined channels **nbrConvertersPerChannel** of **AcqrsD1_configChannelCombination**:

Model	**sFmax vs. nbrConvertersPerChannel**		
	1	**2**	**4**
DC135	¼ × inputFrequency	½ × inputFrequency	
DC140	¼ × inputFrequency	½ × inputFrequency	
DC211/DC211A	2 × inputFrequency		
DC241/DC241A	1 × inputFrequency	2 × inputFrequency	
DC271/DC271A	½ × inputFrequency	1 × inputFrequency	2 × inputFrequency
DC440/DC438/ DP310/DP308	1 × inputFrequency		
DC436/DP306	½ × inputFrequency		
DP214	1 × inputFrequency		
DP235/AP235	½ × inputFrequency	1 × inputFrequency	
DP240/AP240	½ × inputFrequency	1 × inputFrequency	
1st generation	1 × inputFrequency		

Example: When using a DC241 with 2 combined channels, and an external clock frequency of 1800 MHz (= inputFrequency), the possible sampling frequencies are 3.6 GS/s, 1.8 GS/s, 900 MS/s and 450 MS/s.

The ratio of sFmax to inputFrequency can also be learned at run-time by using a call to **AcqrsD1_getInstrumentInfo**(instrumentID," ExtCkRatio", &ratio).

The system computes the required memory overhead (in data samples) on the basis of the current **sampFrequency** and **nbrConvertersPerChannel**. Use the function **AcqrsD1_bestNominalSamples** to obtain the maximum available memory, after setting **these parameters**.

The equivalent of **delayTime** is defined with the value **delayNbrSamples**, which only applies to external clock operation. The actual delay value is easily computed as follows:

delay = (**delayNbrSamples** – **nbrSamples**) / **sampFrequency**

Example: In a 1st generation module with an external clock running at 200 MHz, if you wanted to acquire 2000 data points at 50 MS/s with the trigger point at the end of the first quarter of the time window, you would use the code:

```
AcqrsD1_configMemory(instrumentID,2000,1);
AcqrsD1_configExtClock(instrumentID,1,threshold, 1500, 2.0e+8,
                       5.0e+7);
AcqrsD1_acquire(instrumentID);    // start the acquisition
AcqrsD1_waitForEndOfAcquisition(instrumentID, timeOut);
```

The value of **delayNbrSamples** is 1500 because 500 points need to be acquired *before* and 1500 points *after* the trigger, in order to position the trigger point at the 1st quarter of the time window.

Equivalently, you could have computed the time window to be 2000 x 20 ns = 40 µs. The delay would therefore have to be -10 µs to get the trigger point to the 1st quarter of the time window. Since **nbrSamples** = 2000 and **sampFrequency** = 5.0e+7, you would obtain **delayNbrSamples** = 1500.

Since the sampling frequency is known in this clock mode, through the variable **sampFrequency**, any read functions correctly return the value of the sampling interval.

In addition, if the user-supplied clock frequency **inputFrequency** is > 800 MHz on DC271-FAMILY digitizers or in all cases for 12-bit digitizers, the system correctly measures the value **horPos** and returns it with any waveform read function, such as **AcqrsD1_readData**. Thus, the time position of the sampled data points can be known to within a small fraction of the sampling interval, permitting very precise timing measurements as with the internal clock. However, the digitizer must be calibrated at the external clock frequency in use, whenever **inputFrequency** or **sampFrequency** are changed. Use this code:

```
// We assume that a normal calibration has been done, either
// during initialization, or explicitly
AcqrsD1_configExtClk(.. )    // Set to (cont) Ext Clk
// Make sure to apply the same external frequency as the value
// 'inputFrequency', set in the function call above
AcqrsD1_calibrateEx(instrumentID, 2, 0, 0 );
```

The function **AcqrsD1_calibrateEx** with **calType** = 2 readjusts some timing calibration constants, but does not modify any vertical adjustment values, such as gain or offset.

In first generation digitizers, or when **inputFrequency** is ≤ 800 MHz in the DC271-FAMILY digitizers, the data read functions will return **horPos** = 0.0, equivalent to a timing uncertainty of ± 0.5 samples. For implementation reasons, the acquired waveform in fact has a timing uncertainty that is twice as large, i.e. ± 1samples. In this case, the trigger timestamps of the sequence acquisition mode are not available.

Depending on the ratio of **sampFrequency/inputFrequency** , a waveform is sampled either on negative-going transitions of the external clock signal through the user-defined threshold or, when the ratio is > 1, on both of the transitions.

⚠ ***NOTE:*** *First generation digitizers that have more than one converter/channel (DC240, DP210, and DP211) will generate two data samples for each sampling interval. You must dimension your acquisition and readout for twice the normal amout of data and can then either, drop every other data sample from the record, or average the two data values which could enhance the signal to noise ratio.*

3.12.3. External Clock (Start/Stop)

The *start/stop external clock* mode (**clockType** = 4) permits the application of a (variable) external clock. This can be setup to give bursts during which the frequency is between 10 MHz and 500 MHz. The first sample of each burst may have to be ignored. The waveform is sampled on positive-going transitions of

the external clock signal through the user-defined threshold. Thus, the sampling rate is equal to the input frequency. There is no concept of trigger for this mode and, therefore, all trigger parameters will be ignored. This also means that there is no concept of sequence acquisition. Operation in a channel combined mode is not possible.

In this mode, the horizontal control parameters **sampInterval** and **delayTime** are completely ignored, as well as the value of **delayNbrSamples**. The waveform length is, as usual, controlled by the number of samples in the function **AcqrsD1_configMemory**. Careful synchronization between the function calls to the driver and the generation of the clock burst is required.

There are 2 ways of terminating an acquisition in the start/stop mode:

1. Generate a number of clock transitions that corresponds exactly to the requested number of samples, and stop the acquisition with the function **AcqrsD1_stopAcquisition**. This requires that the host computer obtain some external signal when the clock sequence is terminated.
2. Generate some extra clock transitions, which will fully terminate the acquisition. You can then use the functions **AcqrsD1_acqDone** or **AcqrsD1_waitForEndOfAcquisition** to detect the end of acquisition.

Example for Termination (1): if you wanted to acquire 20 waveforms of 2000 data points each, at a sampling rate of 33.3 MHz, and a time distance of 5 µs between the waveforms, you would use:

```
AcqrsD1_configMemory(instrumentID, 40000, 1);
AcqrsD1_configExtClock(instrumentID, 4, threshold, 0, 33.3e6,
                       33.3e6);
AcqrsD1_acquire(instrumentID);    // start the acquisition
```

➔ Generate 20 bursts of 2000 clock pulses at 33.3 MHz. At the end, you need to inform the host computer to terminate the acquisition and:

```
AcqrsD1_stopAcquisition(instrumentID);
AcqrsD1_readXXXWform(instrumentID, . . .);
```

Note that the sampling rate and the time between bursts have no incidence on the configuration parameters of the digitizer, i.e. they appear nowhere.

Example for Termination (2): if you wanted to acquire 5000 waveforms of 200 data points each, you would write:

```
AcqrsD1_configMemory(instrumentID, 100000, 1);
AcqrsD1_configExtClock(instrumentID, 4, threshold, 0, 0.0, 0.0);
AcqrsD1_acquire(instrumentID);    // start the acquisition
```

➔ Generate 5000 bursts of 200 clock pulses at the required frequency. At the end, you need to generate ≥ 160/320/640 additional clock pulses.

```
AcqrsD1_waitForEndOfAcquisition(instrumentID, timeOut);
AcqrsD1_readXXXWform(instrumentID, . . .);
```

The 320 (640 on 2 GS/s, or 1280 on 4 GS/s digitizers) extra clocks could be generated by 1 or more extra bursts of 200 clock cycles or a special burst. There is no risk of overwriting the earliest data, since the memory is **not** circular in this mode.

Comments valid for both termination mechanisms:

Although the function **AcqrsD1_acquire** sets it to the ready state, the digitizer cannot actually record data while the external clock is idle. The clock burst must start *after* the digitizer has been started, and it must start in a very clean way, i.e. the first pulse must be already well over the threshold and its width must be ≥ 1 ns.

The digitizer sees the multiple clock bursts as a single acquisition. It knows neither the sampling frequency, nor the time difference between the waveforms. It simply acquires a number of data points. In termination mechanism (2) it also records the extraneous points and then stops.

When reading the data, the multiple waveforms appear as a contiguous waveform. The only way of distinguishing one waveform from the other is by counting samples, i.e. the first 2000 samples belong to

the 1st waveform, the next 2000 to the 2nd etc. It is therefore imperative to exactly control the number of clocks in a burst.

⚠ ***NOTE:*** *If the time difference between 2 bursts is > 100 ns, the digitizer tends to drift into saturation, from which it has to recover when the next burst resumes. The first data sample of such a burst is thus invalid. In many cases, this first data sample is sufficiently different from the rest of the waveform that it can serve as a 'segment marker'.*

⚠ ***NOTE:*** *Digitizers that have more than one converter/channel (DC211, DC240, DC241, DP210, DP211, and DP214) will generate two (four-for theDC211) data samples for each sampling interval. You must dimension your acquisition and readout for twice (4x) the normal amout of data and can then either, drop the extra data sample from the record, or average the data values which could enhance the signal to noise ratio.*

3.13. ASBus Operation

The ASBus is intended to synchronize a number of similar CompactPCI modules, in order to make them appear as a single instrument with more channels. After a number of digitizers have been combined with the functions **AcqrsD1_multiInstrAutoDefine** or **AcqrsD1_multiInstrDefine**, each combined instrument can be controlled, when using its **instrumentID**, with the same functions as single digitizers. Acqiris recommends the use of the automatic function, unless you need special control over the order in which the digitizers are numbered within the *MultiInstrument*. Please refer to the function **AcqrsD1_multiInstrAutoDefine**, for details.

If you mix modules (of the same model number) with different memory lengths, you must make sure that you never use more than the shortest memory length available. Otherwise, you will get invalid data on the short memory modules. The automatic function always assigns the clock master role to a module with the shortest memory, with the result that the function **AcqrsD1_configMemory** refuses to accept memory lengths beyond the shortest. When configuring manually, you might want to do the same. Otherwise, you need to explicitly check your requested memory lengths.

3.13.1. Channel Numbering with ASBus

In a *MultiInstrument*, input channels are numbered from 1 to **nbrChannels**. The number of channels can be retrieved with the function call:

```
AcqrsD1_getNbrChannels(instrumentID, &nbrChannels);
```

Channel 1 corresponds to channel 1 of module 0. Channel numbers increase first through module 0, then through modules 1, 2 etc.

For Acqiris CC10x crates when a *MultiInstrument* is defined with **AcqrsD1_multiInstrAutoDefine**, module 0 is always closest to the controller slot (), whereas with **AcqrsD1_multiInstrDefine**, it corresponds to the first module in the initializing list. The Acqiris CC121 crates have a different ordering; please refer to the Acqiris CC121 CompactPCI Crate User Manual.

E.g. when combining 3 DC270 4-channel digitizers, you would use channel number 10 in the function calls **config/get_MultiInput**, **config/get_Vertical** and **readChar/RealWform**, if you wanted to refer to Input 2 of the third DC270.

Channel numbering does not depend on which module is the actual clock or trigger master.

3.13.2. Trigger Source Numbering with ASBus

Acqiris digitizers do not necessarily have as many internal triggers as channel inputs, nor exactly one external trigger. You should retrieve, for every *MultiInstrument*, additional information with the following calls:

```
AcqrsD1_getInstrumentInfo(ID, "NbrInternalTriggers", &nbrIntTrigs);
AcqrsD1_getInstrumentInfo(ID, "NbrExternalTriggers", &nbrExtTrigs);
AcqrsD1_getInstrumentInfo(ID, "NbrModulesInInstrument", &nbrModules);
nbrIntTrigsPerModule = nbrIntTrigs /nbrModules;
nbrExtTrigsPerModule = nbrExtTrigs /nbrModules;
```

In a *MultiInstrument* composed of 4 DC240 (2 channel, 2 GS/s digitizers), you would get 8 internal trigger sources, 4 external trigger sources and 4 for the value of **nbrModules**. Thus, **nbrIntTrigsPerModule** would be 2 and **nbrExtTrigsPerModule** would be 1, as expected for a DC240.

Internal triggers are associated to the input channels, and follow the same numbering rules.

External triggers follow similar rules, i.e. extTrig = 1 corresponds to external trigger 1 of the first module, extTrig = 2 corresponds to external trigger 2 of the first module (if **nbrExtTrigsPerModule** > 1) or to external trigger 1 of the second module etc.

The functions **AcqrsD1_configTrigSource** and **AcqrsD1_getTrigSource** use the explicit trigger channel number, with the internal trigger channel running from 1 to **nbrIntTrigs**, and the external trigger running from –1 to –**nbrExtTrigs**. Note that 0 is an invalid trigger source, resulting in an error code.

The functions **AcqrsD1_configTrigClass** and **AcqrsD1_getTrigClass** encode the trigger source in a 32-bit source pattern:

31	30	20 - 29	16 – 19	5 - 15	4	3	2	1	0
Ext 1	Ext 2	Other Ext Trigs	Module	Other Int Trigs	In 5	In 4	In 3	In 2	In 1

[0]	**IN 1**	Internal trigger channel 1
[1]	**IN 2**	Internal trigger channel 2
[2..4]	**IN 3, IN 4, IN 5**	Internal trigger channels 3, 4, 5
[5..15]	**OTHER INT TRIGS**	Other internal trigger channels within a module, up to 16.
[16..19]	**MODULE**	Module Number, running from 0 to (nbrModules – 1). In single digitizers, this field must be zero. In *MultiInstruments*, the trigger source number must be broken into a module number and a trigger channel number *within* the module.
[20..29]	**OTHER EXT TRIGS**	Other external trigger channels within a module, up to 12.
[30]	**EXT 2**	External trigger channel 2
[31]	**EXT 1**	External trigger channel 1

In future digitizers with trigger pattern capabilities, several trigger bits could be set simultaneously. However, no trigger pattern capabilities *between different modules* can be coded, i.e. only a single module in a MultiInstrument can be the trigger source, although the source in the single module might be a pattern. For these reasons, the module number must be coded explicitly.

To translate a trigger channel number **trigChan** into a trigger source pattern, use the following code:

```
if (trigChan > 0)            // Internal Trigger
{
     long moduleNbr = (trigChan - 1) / nbrIntTrigsPerModule;
     long inputNbr  = (trigChan - 1) % nbrIntTrigsPerModule;
     srcPattern = (moduleNbr<<16) + (0x1<<inputNbr);
}
else if (trigChan < 0)       // External Trigger
{
     trigChan = -trigChan;
     long moduleNbr = (trigChan - 1) / nbrExtTrigsPerModule;
     long inputNbr  = (trigChan - 1) % nbrExtTrigsPerModule;
     srcPattern = (moduleNbr<<16) + (0x80000000>>inputNbr);
}
else
     PROBLEM!
```

Note that **moduleNbr** and **inputNbr** start from 0. An 'industrial strength' implementation should contain some checks on the range of **trigChan** and/or **inputNbr**.

3.14. Special Operating Modes

Some Acqiris digitizers offer alternative operating modes, which are controlled with the function **AcqrsD1_configMode**. The default state of any digitizer is **mode** = 0 and **flags** = 0, corresponding to the *normal* digitizer operation, as described in the other sections of this manual.

3.14.1. Frequency Counter

This is an option available for the DC140 and DC135 digitizers. It is implemented with a signal counter that counts trigger signals from the user-requested channel. A time counter generates the user-programmed aperture time, during which the measurement is performed.

The user-requested signal channel has to be programmed for the expected signal characteristics; the standard config functions should be used to set the full-scale, coupling, offset and trigger threshold. The HF trigger mode may be set by the driver software on the basis of the user-supplied target frequency value. In order to obtain the best results, it is recommended to adjust the full scale and offset so that they span the expected input signal. In addition, the trigger threshold should be set to approximately the center of the signal voltage range. Calls to **AcqrsD1_configVertical** and **AcqrsD1_configTrigClass** are needed. If an external time base reference is desired, **AcqrsD1_configExtClock** should be called. If the totalize in gate functionality is desired the source of the gate must be set with the function **AcqrsD1_configControlIO**, with the parameters *connector* = 1 (I/O A) or 2 (I/O B) and *signal* = 9.

The function **AcqrsD1_configFCounter** sets the parameters specific to the frequency measurements.

```
AcqrsD1_configFCounter(instrumentID, channel, type, targetValue,
                       apertureTime, 0.0, 0);
```

Comments:

- Channel numbers run from 1 to the available number of signal channels in the digitizer.
- The value *type* is 0 for Frequency, 1 for Period, 2 for Totalize by Time and 3 for Totalize by Gate.
- The *targetValue* is an estimator of the expected result. If no estimate is possible, use the value 0.0. This value is only used to activate the HF trigger mode which extends the useable frequency range. By default, the frequency range is extended except:

type	**Measurement**	**Divide by 1**
0	Frequency	if *targetValue* is smaller than 1 kHz (1000.0) and larger than 0.0

1	Period	if *targetValue* is larger than 1ms (0.001)
2	Totalize by Time	always (the HF mode is never used)
3	Totalize by Gate	always (the HF mode is never used)

- The *apertureTime* defines the minimum time for a frequency measurement (it may be longer if the frequency is very low!). In the Totalize by Time mode, the value of *apertureTime* determines the time window during which the input pulses are counted.

The frequency counter mode is set with the function **AcqrsD1_configMode**, with *mode* = 6.

After configuring the instrument parameters, the measurement sequence is started with the function **AcqrsD1_acquire**. This function returns before the measurement is terminated. The user must wait until it is terminated with the functions **AcqrsD1_acqDone** or **AcqrsD1_waitForEndOfAcquisition.** For the case of Totalize by Gate the program must stop the acquisition by making a call to the function **AcqrsD1_stopAcquisition.**

FC results can be readout with the function **AcqrsD1_readFCounter**. The result is always a single double precision number whose units are those appropriate for the *type* of measurement chosen.

3.14.2. 'Start on Trigger'

The 'Start on Trigger' mode begins data recording only upon receipt of a trigger signal, and stops after **nbrSamples** data points are acquired.

It is useful in the special case where the sampling rate is **less** than the maximum possible and where an optimum time correlation between the trigger and the sampling clock is required (typically when averaging waveforms). This mode also requires that the trigger is available *before* the waveform of interest.

In the 'Normal' mode, data recording begins at the time of arming, with the function **AcqrsD1_acquire**. The trigger occurs asynchronously to the sampling clock, and thus will fall randomly anywhere within a sampling interval. When averaging waveforms, this will result in an effective bandwidth reduction since the waveforms are randomly shifted with respect to each other by up to ± ½ sampling interval.

In 'Start on Trigger' mode, the trigger occurs *before* recording starts. It still occurs asynchronously with respect to the internal VCO (which is *always* running). However, if the requested sampling rate is less than the VCO frequency (e.g. 100 MS/s, while the clock runs at 500 MHz), then the time correlation between the trigger and the effective sampling clock is within ± ½ VCO time interval, **not** ± ½ sampling interval. Therefore when averaging, the bandwidth reduction will be less than in the 'normal' mode.

The value **delayTime** in the function **AcqrsD1_configHorizontal** is ignored. As usual, the digitizer requires some memory overhead for additional samples. The function **AcqrsD1_bestNominalSamples** returns the maximum number of available samples.

Use this code to use the 'Start on Trigger' mode:

```
AcqrsD1_configXXX(.. );          // configure other parameters
AcqrsD1_configMode(instrumentID, 0, 0, 1);
AcqrsD1_acquire(instrumentID);
AcqrsD1_waitForEndOfAcquisition(instrumentID, timeout);
// Read out data etc. before calling again "AcqrsD1_acquire"
```

Note that the function **AcqrsD1_acquire** is still needed. However, it behaves somewhat differently in that is does *not* start data recording but waits until a trigger signal is received.

Due to some circuit delays, the waveform recording starts approximately 20ns after the receipt of the trigger signal.

'Start on Trigger' mode is not operational for the 12-bit digitizers.

3.14.3. 'Sequence Wrap'

The normal operation of the digitizer requires that it stop recording waveforms when the pre-defined number of segments has been acquired. Thus, **nbrSegments** triggers are needed to acquire the requested

number of segments into the same number of different memory sections. After the acquisition has terminated, all of the waveform segments are finally available for readout.

The 'Sequence Wrap' mode also pre-defines the desired number of different memory sections, but it permits a larger number of triggers. After the first **nbrSegments** waveform segments are acquired, the digitizer 'wraps' around to the first memory segment and keeps on recording waveforms. This sequence can go on indefinitely, since no hardware condition will stop it. The only way to terminate this infinite loop is to stop it with the function **AcqrsD1_stopAcquisition**.

This mode is useful when only the last N out of many occurrences of a signal are of interest. E.g. if you search for a rare event out of many occurrences, and you only can determine its interest *after* the event has occurred, then the 'Sequence Wrap' mode is applicable.

While this mode even 'works' for **nbrSegments** = 1, in practice the value of **nbrSegments** should be at least 2. It is important to note that after the acquisition of a segment, the digitizer automatically advances to the next memory section and immediately (with a dead time of ~ 1 µs) starts recording into it. Thus, the very last memory segment will necessarily contain uninteresting data, since it will not be stopped with a trigger, but be terminated with the software command **AcqrsD1_stopAcquisition**.

Use this code to use the 'Sequence Wrap' mode:

```
AcqrsD1_configXXX(.. );          // configure other parameters
AcqrsD1_configMode(instrumentID, 0, 0, 2);
AcqrsD1_acquire(instrumentID);
.
.
AcqrsD1_stopAcquisition(instrumentID);
```

The time at which the sequence is terminated with the function **AcqrsD1_stopAcquisition** depends on an external event, e.g. operator intervention.

When reading the segments, the segment number should take on the values 0,…(**nbrSegments**-1). They correspond to the memory section numbers in the digitizer, **not** the time order of the acquired segments.

Example: if **nbrSegments** = 8, the time order of the acquired segments might be (depending on when the sequence was stopped) 5, 6, 7, 0, 1, 2, 3, 4. Here, the 'oldest good' segment is the 5th segment, followed by the 6th, 7th, 8th, 1st etc. The 'youngest' useful segment is the 3rdone, while the 4th segment corresponds to the segment that was being recorded when the stop-command was received. The 4th segment (in this example) typically does not contain any useful data.

3.15. Readout of Battery Backed-up Memories

Acqiris digitizers with the battery back-up option permit retaining acquired waveforms during periods of time when the power might be interrupted.

3.15.1. Preparations before Power-Off

A digitizer only remembers the digitized data array, but not all of the parameters that are needed to interpret the waveform. These parameters are normally retained in the driver, but are typically lost when power is lost or when the controlling application is terminated.

It is therefore necessary for the application to transfer the relevant parameters **before** power-off, typically to a disk, in such a way that they are again available when restarting the application after power is restored. The following parameters must be transferred to persistent storage, for each instrument:

- Parameters of **AcqrsD1_configHorizontal**, i.e. sampling interval and delay
- Parameters of **AcqrsD1_configMemory**, i.e. number of samples and number of segments
- Parameters of **AcqrsD1_configVertical**, i.e. Full Scale and offset (the coupling is not relevant!), for **each** channel of interest
- Two calibrated delay parameters, as obtained with the function calls

```
Double delayOffset, delayScale;
AcqrsD1_getInstrumentInfo(ID, "DelayOffset", &delayOffset);
AcqrsD1_getInstrumentInfo(ID, "DelayScale", &delayScale);
```

The functions **AcqrsD1_getInstrumentInfo** should be called just before the start of the acquisition, i.e. before calling **AcqrsD1_acquire**, but after all **AcqrsD1_config…** functions.

3.15.2. Recovery after Power-Off

In order to read a battery backed-up waveform, you need to execute a special sequence of initialization functions

1. Initialize the digitizer with the following function call:

   ```
   AcqrsD1_InitWithOptions(resourceName, false, false, "CAL=FALSE",
                                                       &instrumentID);
   ```

   It is important to specify "CAL=FALSE" to prevent any calibration in the digitizer, which would destroy any retained data.

2. Call the functions **AcqrsD1_configHorizontal**, **AcqrsD1_configMemory**, **AcqrsD1_configVertical** (for each channel!) with the same parameters that were used in the original acquisition.

3. Call the function **AcqrsD1_restoreInternalRegisters**, with the parameters *delayOffset* and *delayScale*. In case these parameters are not available (e.g. due to earlier software versions), you should use the values –20.0e-9 for *delayOffset* and 5.0e-12 for *delayScale*.

4. Call the function **AcqrsD1_readData** to read the battery backed-up data.

Failing to restore the originally used digitizer parameters may result in erroneous data. After data recovery, and before using the digitizer for any new acquisitions, don't forget to calibrate the instrument with the function **AcqrsD1_configTrigSource**.

3.16. Reading the Digitizer Temperature

The temperature of a digitizer can be obtained with the following code:

```
long temperature; // will be in degrees C
AcqrsD1_getInstrumentInfo(ID, "Temperature", &temperature);
```

When multiple digitizers are combined via ASBus to a MultiInstrument, use the strings `"Temperature 0", "Temperature 1"`… to refer to the individual modules.

⚠ ***NOTE:*** *The returned temperature value corresponds to the ambient temperature on the main printed-circuit board, typically near the timebase circuit. It cannot represent all possible temperature values that are present on the circuit. Values ≥ 60 °C indicate that the circuit is near its operational limit, and a cooling failure occurred or better cooling should be installed.*

Acqiris recommends keeping the temperature as low as possible, since a 10°C reduction in circuit temperature is expected to improve the mean-time-between-failures (MTBF) by a factor of 2.

Acqiris also recommends reading the temperature when the digitizer is stopped. The read operation may generate small signal perturbations through cross talk, if it is executed while an acquisition is in progress.

4. Attributes

Module settings can be read and/or changed through the use of attributes. The attribute interface to the driver can be used with AcqirisMAQS, the MATLAB interface, and the SP201 Software Development Kit. This section of the manual will give details on all of the available attributes.

The full name of an attribute is often composed of a prefix before the name of the simple attribute itself. In other words,

full_attribute_name := [<*prefix*>]*attribute*

prefix := *channel_prefix*|*external_prefix*

channel_prefix := "ch1"|"ch2"|"ch3"|"ch4"|"ch5"| … |"chnn"

where nn is the last channel number of the instrument.

external_prefix := "ext1"|"ext2"| … |"extm"

where m is the last external trigger channel of the instrument. The number of channels can be found from the *channels* keyword of the instrument identity. Similarly, the number of external triggers can be found from the *confExtTr* keyword.

Examples:

- Full scale for channel 2: <ch2>chFullScale
- Number of samples: samples
- External trigger slope for single module (noASBUs): <ext1>trSlope

Attributes can either be modifiable or read-only. This is shown by the Access characteristic that is either RW (read-write) or RO (read-only), respectively. Only the RW-client of an instrument can actually write/modify an RW attribute.

The type of an attribute shows what kind of value will be employed when an attribute is queried or modified. Although the argument is always a string, the string will be a representation of an integer, a double precision real, or a string. It also indicates whether the range of values will be described as discrete or not. If the range is discrete the allowed values will be enumerated explicitly; otherwise the full range will be described as a collection of triplets, {low value, high value, increment}. In the case where the increment is non-zero, a triplet defines allowed values of the attribute as follows:

low value, low value + increment, low value + 2 * increment, …, high value

If the increment is 0 all values between the low and high values are allowed.

4.1. Attribute dependencies and coerced values

Changing certain attribute values can affect the ranges of other attributes. This can cause the value of the dependent attribute to change. For example, changing the **chFullScale** of a channel can change the ranges for **chOffset** and **trThres1**. Similarly, requested attribute values will not always be available. In order to verify that all changes are consistent with your expectations you can always check final attribute values with the GetAttributeValue method.

4.2. Functional grouping of attributes

Channel (vertical) settings

chAttenuation	chBandwidthLimit	chCoupling
chFullScale	chImpedance	chInput
chOffset	chScale2ndFactor	chScaleFactor
chScaleOffset	chUnits	

Horizontal timebase settings

chTime2ndOffset	chTime3rdOffset	chTimeOffset
ckDelay	ckThreshLvl	ckType
extCkRatio	extInputFreq	extSamplingRate
extSparsing	samples	samplingFreq
samplingIntv	segments	timeWindow
trDelay		

Trigger settings

chBandwidthLimit	chFullScale	trClass
trCoupling	trRelThres1	trRelThres2
trSlope	trSrcEnabled	trThres1
TrThres2	trThresUnit	trTVField
trTVLine	trTVStandard	

Control IO settings

ctrlIOASignal	ctrlIOBSignal	ctrlIOPxiRef10MHz
ctrlIOPxiStar	ctrlIOTrOutLevel	ctrlIOTrOutResync

Others

channelEnable	channelName	channelUse
convertersPerChan	input	name

Averager settings

axDitherRange	axNbrRoundRobins	axNbrWaveforms
axP1Signal	axP2Signal	axSamples
axSegments	axStartDelay	axStartVetoEnable
axStopDelay	axTrigAlways	axTrigResync
chAxInvertData	chAxNoiseBase	chAxNoiseBaseEnable
chAxTDCHistoEnable	chAxTDCHistoIncr	chAxTDCHistoMinTOT
chAxThreshold	chAxThresholdEnable	

Not applicable

chComment	chSetupName	instrComment

instrSetupName	mode	modeOption
tbComment	trComment	trSrcComment

4.3. Traditional API grouping of attributes

Old API Uage (AcqrsD1…)	**Attribute equivalent**		
_configAvgConfig	axDitherRange axP1Signal axSegments axStopDelay chAxInvertData chAxTDCHistoEnable chAxThreshold	axNbrRoundRobins axP2Signal axStartDelay axTrigAlways chAxNoiseBase chAxTDCHistoIncr chAxThresholdEnable	axNbrWaveforms axSamples axStartVetoEnable axTrigResync chAxNoiseBaseEnable chAxTDCHistoMinTOT
_configChannelCombination	input		
_configControlIO	ctrlIOASignal, ctrlIOPxiStar,	ctrlIOBSignal, ctrlIOTrOutLevel,	ctrlIOPxiRef10MHz, ctrlIOTrOutResync
_configExtClock	ckDelay, extInputFreq,	ckThreshLvl, extSamplingRate	ckType,
_configHorizontal	trDelay,	samplingFreq,	samplingIntv
_configMemory	samples	segments	
_configMultInput	chInput		
_configTrigClass	trClass,	trSrcEnabled	
_configTrigSource	trCoupling, trSlope,	trRelThres1, trThres1	trRelThres2, trThres2
_configTrigTV	trTVField,	trTVLine	trTVStd
_configVertical	chCoupling, chImpedance,	chBandwidthLimit, chOffset	chFullScale,
_getChannelCombination	convertersPerChan,	input	
_getControlIO	ctrlIOASignal, ctrlIOPxiStar,	ctrlIOBSignal, ctrlIOTrOutLevel,	ctrlIOPxiRef10MHz, ctrlIOTrOutResync
_getExtClock	ckDelay, extInputFreq,	ckThreshLvl, extSamplingRate	ckType,
_getHorizontal	trDelay,	samplingFreq,	samplingIntv
_getMemory	samples	segments	
_getMultInput	chInput		
_getTrigClass	trClass,	trSrcEnabled	

_getTrigSource	trCoupling, trSlope,	trRelThres1, trThres1	trRelThres2, trThres2
_getTrigTV	trTVField,	trTVLine	trTVStd
_getVertical	chCoupling, chImpedance,	chBandwidthLimit, chOffset	chFullScale,

4.4. Detailed description of attributes (Alphabetically ordered)

axDitherRange

Purpose

Defines the range of offset dithering, in ADC LSB's

Prefixes	Access	Type	Values
	RW	integer	[0,15] in LSB's

Models

AP235 and AP240 with Averager option, AP100,AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "DitherRange", &vdither)

Discussion

The offset is dithered over the range [-v, + v] in steps of ~1/8 LSB.

axNbrRoundRobins

Purpose

Number of times to perform the full segment cycle during data accumulation

Prefixes	Access	Type	Values
	RW	integer	1 to 65536

Models

AP235 and AP240 with Averager option

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "NbrRoundRobins", &robin)

Discussion

axNbrWaveforms

Purpose

Number of waveforms to average before going to next segment

Prefixes	Access	Type	Values
	RW	integer	1 to 65535

Models

AP235 and AP240 with Averager option, AP100,AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "NbrWaveforms", &triggers)

Discussion

axP1Signal

Purpose

Define how the P1 Control signal should be used.

Prefixes	Access	Type	Values
	RW	String	"Disabled", "In-AddSubCh1", "In-AddSubCh2", "In-AddSubCh1Ch2", "In-AvgrTrigEnable", "In-AvgrStartVetoEnable", "Out-Average"

Models

AP235 and AP240 with Averager option

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "P1Control", &function)

Discussion

The "In-AddSub…" choices allow the control of real time add/subtract functionality for either or both of the channels. The " In-AvgrStartVetoEnable " configures the signal as a trigger enable.

If P1Signal and/or P2 Signal are enabled for the Add/Subtract mode then the data will be added if the signal, or the or of both signals, is in the high state. The same rule holds if they are used for trigger enable.

The "Out-Average" signal goes high after the first trigger is accepted for an average and drops back down when the last trigger's acquition is complete.

axP2Signal

Purpose

Define how the P2 Control signal should be used.

Prefixes	Access	Type	Values
	RW	string	"Disabled", "In-AddSubCh1", "In-AddSubCh2", "In-AddSubCh1Ch2", "In-AvgrTrigEnable", "In-AvgrStartVetoEnable", "Out-Average"

Models

AP235 and AP240 with Averager option

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "P2Control", &function)

Discussion

The "In-AddSub…" choices allow the control of real time add/subtract functionality for either or both of the channels. The "In-Average" configures the signal as a trigger enable.

If P1Signal and/or P2 Signal are enabled for the Add/Subtract mode then the data will be added if the signal, or the or of both signals, is in the high state. The same rule holds if they are used for trigger enable.

The "Out-Average" signal goes high after the first trigger is accepted for an average and drops back down when the last trigger's acquition is complete.

axSamples

Purpose

Number of data samples per waveform segment in Averager mode

Prefixes	Access	Type	Values
	RW	integer	

Models

AP235 and AP240 with Averager option, AP100,AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "NbrSamples", &samples)

Discussion

May assume values between 16 or 32 and the available memory length, in multiples of 16 (32). The granularity is 16 for the AP100/AP101 and the AP240/AP235 in Dual-Channel mode and 32 for the AP200/AP201 and the AP240/AP235 in Single-Channel mode.

axSegments

Purpose

Number of waveform segments to acquire

Prefixes	Access	Type	Values
	RW	integer	

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "NbrSegments", &segments)

Discussion

May assume values between 1 and 8192.

axStartDelay

Purpose

Start delay in samples

Prefixes	Access	Type	Values
	RW	integer	

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "StartDelay", &start)

Discussion

May assume values between 0 and 33554400(16777216) in steps of 16 (32) as explained below. The limit is StepSize*(1024*1024-1). The granularity is 16 for the AP100/AP101 and the AP240/AP235 in Dual-Channel mode and 32 for the AP200/AP201 and the AP240/AP235 in Single-Channel mode.

axStartVetoEnable

Purpose

Select the functionality for the Control I/O Trigger enable Input

Prefixes	Access	Type	Values
	RW	string	"Trigger enable", "Start Veto"

Models

AP100, AP200 ONLY

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "StartVetoEnable", &start)

Discussion

This must be used in conjunction with **ctrlIOASignal** or **ctrlIOBSignal** being set to

"Trigger enable".

If set to "Start Veto" an Average Accumulation will only start after the computer starts the acquisition if the signal is high or if it goes high.

axStopDelay

Purpose

Stop delay in samples

Prefixes	Access	Type	Values
	RW	integer	

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "StopDelay", &stop)

Discussion

May assume values between 0 and 2097120(1048560) in steps of of 16 (32) as explained below. The limit is StepSize*(64*1024-1). The granularity is 16 for the AP100/AP101 and the AP240/AP235 in Dual-Channel mode and 32 for the AP200/AP201 and the AP240/AP235 in Single-Channel mode.

axTrigAlways

Purpose

Control whether the trigger out signal should be generated only for accepted triggers or for all of them.

Prefixes	Access	Type	Values
	RW	string	"Off" or "On"

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "TrigAlways", &always)

Discussion

axTrigResync

Purpose

Controls whether the trigger should be used to resynchronize the sampling clock.

Prefixes	Access	Type	Values
	RW	string	"NoResync", "Resync", or "FreeRun"

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "TrigResync", &sync)

Discussion

The "TrigResync" values "NoResync" and "Resync" require a valid trigger, while "FreeRun" requires no trigger (useful for background acquisition).

chAxInvertData

Purpose

Indicate whether the data of the channel should be inverted

Prefixes	Access	Type	Values
channel	RW	string	"Off" or "On"

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "InvertData", &invert)

Discussion

If the data is inverted the most significant bit of the raw data will be flipped; the threshold will select data values with voltages below the desired value.

chAxNoiseBase

Purpose

Value in Volts of the value to be added in Noise Supressed Averaging

Prefixes	Access	Type	Values
channel	RW	double	In Volts

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "NoiseBase", &base)

Discussion

The value will be transformed into the ADC raw data equivalent of the voltage value.

chAxNoiseBaseEnable

Purpose

Allows the Noise Base value to be added to the data in Noise Supressed Averaging

Prefixes	Access	Type	Values
channel	RW	string	"Off" or "On"

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "NoiseBaseEnable", &enable)

Discussion

chAxTDCHistoEnable

Purpose

Enable the simple TDC mode for the channel

Prefixes	Access	Type	Values
channel	RW	string	"Off" or "On"

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "HistoTDCEnable", &enable)

Discussion

chAxTDCHistoIncr

Purpose

Define the increment to be used in accumulating a simple TDC histogram

Prefixes	Access	Type	Values
channel	RW	Integer	1 or 2

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "TdcHistogramIncrement", &style)

Discussion

1 means increment by 1,
2 means increment by the ADCvalue – NoiseBase equivalent.

.

chAxTDCHistoMinTOT

Purpose

Controls the desired minimum width of a peak in the waveform

Prefixes	Access	Type	Values
channel	RW	integer	1, 2, 3, or 4

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "TdcMinTOT", &width)

Discussion

A peak is accepted if there are at least n consecutive data samples above the Threshold. For SimpleTDC mode ONLY.

chAxThreshold

Purpose

Value in Volts of the threshold for Noise Supressed Averaging

Prefixes	Access	Type	Values
channel	RW	double	In Volts

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "Threshold", &thresh)

Discussion

chAxThresholdEnable

Purpose

Control whether the threshold test will be applied to the data of the channel before accumulation

Prefixes	Access	Type	Values
channel	RW	string	"Off" or "On"

Models

AP235 and AP240 with Averager option, AP100, AP200

Traditional API equivalent

AcqrsD1_configAvgConfig(id, channel, "ThresholdEnable", &enable)

Discussion

channelUse

Purpose

Allows the RW-client to activate/de-activate a channel

Prefixes	Access	Type	Values
channel	RW	string	"on" or "off"

Models

ALL

Traditional API equivalent

NONE

Discussion

Deactivation of a channel can speed-up clients and the server. Deactivated channels do not have to have their data read out nor transmitted over the network.

chAttenuation

Purpose

Functions as an additional multiplier for the conversion of raw ADC values to volts or other user units

Prefixes	Access	Type	Values
channel	RW	double	In dB

Models

ALL

Traditional API equivalent

NONE

Discussion

The raw integer data will be converted into a real value using several factors. This one is in dB.

chBandwidthLimit

Purpose

Control the setting of the bandwidth limiter of a channel or external trigger input

Prefixes	Access	Type	Values
channel & ext	RW	double, discrete	In Hz (0.0 = off)

Models

ALL

Traditional API equivalent

```
AcqrsD1_configVertical(id, …, bandwidth)
AcqrsD1_getVertical(id, …, &bandwidth)
```

Discussion

The bandwidth is either given as 0.0 for no limit or a frequency value corresponding to the upper limit.

chCoupling

Purpose

Control the coupling of a channel or external trigger

Prefixes	Access	Type	Values
channel & ext	RW	string	"AC","DC"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configVertical(id, …, coupling, …)
AcqrsD1_getVertical(id, …, &coupling, …)
```

Discussion

The chCoupling is only part of the coupling of the old API. chImpedance is also needed.

chFullScale

Purpose

Control the full scale of a channel

Prefixes	Access	Type	Values
channel & ext	RW	double, discrete	In V

Models

ALL

Traditional API equivalent

```
AcqrsD1_configVertical(id, channel, fullScale, …)
AcqrsD1_getVertical(id, channel, &fullScale, …)
```

Discussion

For a channel, the gain/LSB is the full scale divided by the number of bits in the output data representation. This is not always the same as the ADC gain that is calculated using the actual number of bits of the ADC. For an external trigger, this attribute is used to control the full scale range of the allowed trigger level. Studying the current range of the attribute may be needed to understand all of the possible values.

chImpedance

Purpose

Control the input impedance of a channel

Prefixes	Access	Type	Values
channel & ext	RW	string	"50","1M","Gnd"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configVertical(id, …, coupling, …)
AcqrsD1_getVertical(id, …, &coupling, …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

chInput

Purpose

Control the choice of input BNC for some single channel units

Prefixes	Access	Type	Values
channel	RW	string	"A" or "B"

Models

DP111, DP211

Traditional API equivalent

```
AcqrsD1_configMultInput(id, channel, input)
AcqrsD1_getMultInput(id, channel, &input)
```

Discussion

This command is only of interest for single channel units allowing the choice of A or B inputs.

chOffset

Purpose

Control the DC offset of a channel

Prefixes	Access	Type	Values
channel	RW	double	In V

Models

ALL

Traditional API equivalent

```
AcqrsD1_configVertical(id, channel, …, offset,…)
AcqrsD1_getVertical(id, channel, …, &offset,…)
```

Discussion

The mid-range ADC value corresponds to the offset. This is the opposite of the convention in the traditional API.

chScale2ndFactor

Purpose

Functions as an additional multiplier for the conversion of raw ADC values to volts or other user units

Prefixes	Access	Type	Values
channel	RW	double	

Models

ALL

Traditional API equivalent

NONE

Discussion

See the MATLAB discussion in the Programmer's Guide section 2.5 or section 6.2.4 of the SP201 Manual.

chScaleFactor

Purpose

Functions as an additional multiplier for the conversion of raw ADC values to volts or other user units

Prefixes	Access	Type	Values
channel	RW	double	

Models

ALL

Traditional API equivalent

NONE

Discussion

See the MATLAB discussion in the Programmer's Guide section 2.5 or section 6.2.4 of the SP201 Manual.

chScaleOffset

Purpose

Functions as an additive constant for the conversion of raw ADC values to volts or other user units

Prefixes	Access	Type	Values
channel	RW	double	

Models

ALL

Traditional API equivalent

NONE

Discussion

See the MATLAB discussion in the Programmer's Guide section 2.5 or section 6.2.4 of the SP201 Manual.

chTime2ndOffset

Purpose

Functions as an additive constant to adjust the time of the data points of an acquisition

Prefixes	Access	Type	Values
	RW	double	In seconds

Models

ALL

Traditional API equivalent

NONE

Discussion

See the MATLAB discussion in the Programmer's Guide section 2.5 or section 6.2.4 of the SP201 Manual.

chTime3rdOffset

Purpose

Functions as an additive constant to adjust the time of the data points of an acquisition

Prefixes	Access	Type	Values
	RW	double	In seconds

Models

ALL

Traditional API equivalent

NONE

Discussion

See the MATLAB discussion in the Programmer's Guide section 2.5 or section 6.2.4 of the SP201 Manual.

chTimeOffset

Purpose

Functions as an additive constant to adjust the time of the data points of an acquisition

Prefixes	Access	Type	Values
	RW	double	In seconds

Models

ALL

Traditional API equivalent

NONE

Discussion

See the MATLAB discussion in the Programmer's Guide section 2.5 or section 6.2.4 of the SP201 Manual.

chUnits

Purpose

Allows the RW-client to set the vertical units string for data values of a channel for the use of all clients

Prefixes	Access	Type	Values
channel	RW	string	any string

Models

ALL

Traditional API equivalent

NONE

Discussion

Digitizer data in volts can be subject to a linear transformation, which also changes the units of the result. This attribute stores the string associated with the new unit so that it is available to all client applications. See the discussion in section 2.5 ***MATLAB*** or section 6.2.4 of the SP201 Manual.

ckDelay

Purpose

Control the total number of samples digitized in a Continuous External clock acquisition

Prefixes	Access	Type	Values
	RW	integer	

Models

ALL

Traditional API equivalent

```
AcqrsD1_configExtClock(id, …, delayNbrSamples, …)
AcqrsD1_getExtClock(id, …, &delayNbrSamples, …)
```

Discussion

See section 3.11 of the Programmer's Reference manual.

ckThreshLvl

Purpose

Control the threshold level for External clock or Reference operation

Prefixes	Access	Type	Values
	RW	double	

Models

ALL

Traditional API equivalent

```
AcqrsD1_configExtClock(id, …, inputThreshold, …)
AcqrsD1_getExtClock(id, …, &inputThreshold, …)
```

Discussion

See section 3.11 of the Programmer's Reference manual.

ckType

Purpose

Control the type of External clock or Reference operation

Prefixes	Access	Type	Values
	RW	string	"Internal", "ExtCkContinuous", "ExtRef10MHz", "ExtCkStartStop"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configExtClock(id, clockType, …)
AcqrsD1_getExtClock(id, &clockType, …)
```

Discussion

See section 3.11, External Clock and Reference, of the Programmer's Reference manual.

convertersPerChan

Purpose

Returns the number of ADC converters used for each channel

Prefixes	Access	Type	Values
	RO	integer	1,2,4

Models

DC135, DC140, DC241, DC241A, DC271, DC271A, DP235, DP240, AP235, AP240

Traditional API equivalent

```
AcqrsD1_getChannelCombination(id, &nbrConvertersPerChannel,…)
```

Discussion

Use **input** to control both this number and the choice of active channels.

ctrlIOASignal

Purpose

Control the functionality of the CtrlI/O A connector

Prefixes	Access	Type	Values
	RW	string	

Models

DC271-FAMILY, 12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configControlIO(id, 1, …)
AcqrsD1_getControlIO(id, 1, …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

ctrlIOBSignal

Purpose

Control the functionality of the CtrlI/O B connector

Prefixes	Access	Type	Values
	RW	string	

Models

DC271-FAMILY, 12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configControlIO(id, 2, …)
AcqrsD1_getControlIO(id, 2, …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

ctrlIOPxiRef10MHz

Purpose

Control the use of the PXI Reference clock

Prefixes	Access	Type	Values
	RW	string	"Off", "On"

Models

DC135, DC140, DC211, DC211A, DC241, DC241A, DC271, DC271A

Traditional API equivalent

```
AcqrsD1_configControlIO(id, 11, …)
AcqrsD1_getControlIO(id, 11, …)
```

Discussion

The value “On” will set the instrument to use the PXI 10 MHz reference as the internal clock reference.

ctrlIOPxiStar

Purpose

Control the use of the PXI Star trigger

Prefixes	Access	Type	Values
	RW	string	"Off", "TrigIn", "TrigOut"

Models

DC135, DC140, DC211, DC211A, DC241, DC241A, DC271, DC271A

Traditional API equivalent

```
AcqrsD1_configControlIO(id, 12, …)
AcqrsD1_getControlIO(id, 12, …)
```

Discussion

The value “TrigOut” should be chosen when the instrument is the PXI Star Trigger generator.

To trigger on the PXI Star signal, the value “TrigIn” should be chosen.

ctrlIOTrOutLevel

Purpose

Control the offset of the Trigger Out signal

Prefixes	Access	Type	Values
	RW	double	In mV

Models

DC271-FAMILY, 12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configControlIO(id, 9, signal …)
AcqrsD1_getControlIO(id, 9, &signal …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

ctrlIOTrOutResync

Purpose

Control the synchronization of the trigger out signal with respect to the sampling clock

Prefixes	Access	Type	Values
	RW	string	"Off", "On"

Models

DC271-FAMILY, 12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configControlIO(id, 9, …, qualifier1, …)
AcqrsD1_getControlIO(id, 9, …, &qualifier1, …)
```

Discussion

Use this attribute to control if the trigger output is resynchronized to the clock or maintains a precise timing relation to the trigger input.

extCkRatio

Purpose

Give the ratio of the maximum external (continuous) clock sampling rate to the external clock input frequency

Prefixes	Access	Type	Values
	RO	double	1.0, 0.5

Models

ALL

Traditional API equivalent

NONE

Discussion

This number can depend on the model and the **convertersPerChan** value.

extInputFreq

Purpose

Inform the driver of the external (continuous) clock frequency

Prefixes	Access	Type	Values
	RW	double	In Hz

Models

ALL

Traditional API equivalent

```
AcqrsD1_configExtClock(id, …, inputFrequency, …)
AcqrsD1_getExtClock(id, …, &inputFrequency, …)
```

Discussion

See section 3.11, External Clock and Reference, of the Programmer's Reference manual.

extSamplingRate

Purpose

Control the desired sampling rate when in external (continuous) clock mode

Prefixes	Access	Type	Values
	RW	double	In Hz

Models

ALL

Traditional API equivalent

```
AcqrsD1_configExtClock(id, …, sampFrequency)
AcqrsD1_getExtClock(id, …, &sampFrequency)
```

Discussion

See section 3.11, External Clock and Reference, of the Programmer's Reference manual.

extSparsing

Purpose

Alternate to control the desired sampling rate when in external (continuous) clock mode

Prefixes	Access	Type	Values
	RW	double	

Models

ALL

Traditional API equivalent

```
AcqrsD1_configExtClock(id, …, sampFrequency)
AcqrsD1_getExtClock(id, …, &sampFrequency)
```

Discussion

The values are such that extSamplingRate = extCkRatio * extInputFreq / extSparsing.

See also section 3.11, External Clock and Reference, of the Programmer's Reference manual.

input

Purpose

Control the active channels and number of converters per channel

Prefixes	Access	Type	Values
	RW	string	discrete

Models

DC135, DC140, DC241, DC241A, DC271, DC271A, DP235, DP240, AP235, AP240

Traditional API equivalent

```
AcqrsD1_configChannelCombination(id, …, usedChannels)
AcqrsD1_getChannelCombination(id, …, &usedChannels)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

name

Purpose

Allows the RW-client to set the name of the instrument for the use of all clients

Prefixes	Access	Type	Values
	RW	string	any string

Models

ALL

Traditional API equivalent

NONE

Discussion

The RW-client can set a name that can then be known by all the other clients, e.g. for better (human) identification of the instrument.

samples

Purpose

Control the number of valid data points (per segment) in an acquisition

Prefixes	Access	Type	Values
	RW	integer	

Models

ALL

Traditional API equivalent

```
AcqrsD1_configMemory(id, nbrSamples, …)
AcqrsD1_configHorizontal(id, &nbrSamples, …)
```

Discussion

The maximum number of samples depends on the current number of segments in a sequence acquisition. Studying the current range of the attribute may be needed to understand the current range.

samplingFreq

Purpose

Control the time between two data points in an acquisition

Prefixes	Access	Type	Values
	RW	double	In Hz

Models

ALL

Traditional API equivalent

```
AcqrsD1_configHorizontal(id, delayTime, sampInterval)
AcqrsD1_configHorizontal(id, &delayTime, &sampInterval)
```

Discussion

The samplingFreq is the inverse of the sampInterval. Studying the current range of the attribute may be needed to understand all of the possible values.

samplingIntv

Purpose

Control the time between two data points in an acquisition

Prefixes	Access	Type	Values
	RW	double	In seconds

Models

ALL

Traditional API equivalent

```
AcqrsD1_configHorizontal(id, delayTime, sampInterval)
AcqrsD1_configHorizontal(id, &delayTime, &sampInterval)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

segments

Purpose

Control the number of segments in an acquisition

Prefixes	Access	Type	Values
	RW	integer	

Models

ALL

Traditional API equivalent

```
AcqrsD1_configMemory(id, …, nbrSegments)
AcqrsD1_getMemory(id, …, &nbrSegments)
```

Discussion

The number of segments chosen may coerce the number of samples per segment in the acquisition.

timeWindow

Purpose

Control the total time spanned by a segment in an acquisition.

Prefixes	Access	Type	Values
	RW	double	In seconds

Models

ALL

Traditional API equivalent

NONE

Discussion

timeWindow = samples * samplingIntv

trClass

Purpose

Control the type of trigger

Prefixes	Access	Type	Values
	RW	string	"Edge", "TV"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configTrigClass(id, trigClass, …)
AcqrsD1_getTrigClass(id, &trigClass, …)
```

Discussion

Studying the current range of the attribute may be needed to understand the current range.

trCoupling

Purpose

Control the trigger coupling of a trigger source signal

Prefixes	Access	Type	Values
channel	RW	string	"DC", "AC", "HFRej"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configTrigSource(id, channel, trigCoupling, …)
AcqrsD1_getTrigSource(id, channel, &trigCoupling, …)
```

Discussion

This attribute controls the coupling of the trigger source signal to the trigger discriminator.

Studying the current range of the attribute may be needed to understand the current range.

trDelay

Purpose

Control the pre- or post-trigger delay time for an acquisition

Prefixes	Access	Type	Values
	RW	double	In seconds

Models

ALL

Traditional API equivalent

```
AcqrsD1_configHorizontal(id, delayTime, …)
AcqrsD1_getHorizontal(id, &delayTime, …)
```

Discussion

Studying the current range of the attribute may be needed to understand the current range.

trRelThres1

Purpose

Control the trigger discriminator level for triggering

Prefixes	Access	Type	Values
channel	RW	double	in % of full scale

Models

ALL

Traditional API equivalent

```
AcqrsD1_configTrigSource(id, channel, …, trigLevel1, …)
AcqrsD1_getTrigSource(id, &delayTime, …, &trigLevel1, …)
```

Discussion

The values for this attribute must be given in %FS, and where 0.0 corresponds to the middle of the input range.

The trigger discriminator level can also be set in volts. This is done with **trThres1**.

trRelThres2

Purpose

Control the second trigger discriminator level for window triggering

Prefixes	Access	Type	Values
channel	RW	double	in % of full scale

Models

DC271-FAMILY

Traditional API equivalent

```
AcqrsD1_configTrigSource(id, channel, …, trigLevel2)
AcqrsD1_getTrigSource(id, &delayTime, …, &trigLevel2)
```

Discussion

The values for this attribute must be given in %FS, and where 0.0 corresponds to the middle of the input range.

The second trigger discriminator level can also be set in volts. This is done with **trThres2**.

trSlope

Purpose

Control the slope or type of the transition to be used for the trigger

Prefixes	Access	Type	Values
channel & ext	RW	string	"Pos", "Neg", "WindowIn", "WindowOut", "HFDiv4"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configTrigSource(id, channel, …, trSlope, …)
AcqrsD1_getTrigSource(id, &delayTime, …, &trSlope, …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

trSrcEnabled

Purpose

Control which input should be used for the trigger

Prefixes	Access	Type	Values
channel & ext	RW	string	"Off", "On"

Models

ALL

Traditional API equivalent

```
AcqrsD1_configTrigClass (id, …, sourcePattern, …)
AcqrsD1_getTrigClass(id, …, &sourcePattern, …)
```

Discussion

The current trigger source can only be changed by turning “On” another source.

trThres1

Purpose

Control the trigger discriminator level for triggering

Prefixes	Access	Type	Values
channel & ext	RW	double	In V

Models

ALL

Traditional API equivalent

```
AcqrsD1_configTrigSource(id, channel, …, trigLevel1, …)
AcqrsD1_getTrigSource(id, &delayTime, …, &trigLevel1, …)
```

Discussion

For channels, the trigger discriminator level can also be controlled relative to the current input range (as defined by the full scale and offset). This is done with **trRelThres1**, which takes values in %FS, and where 0.0 corresponds to the middle of the input range.

trThres2

Purpose

Control the second trigger discriminator level for window triggering

Prefixes	Access	Type	Values
channel & ext	RW	double	In V

Models

DC271-FAMILY

Traditional API equivalent

```
AcqrsD1_configTrigSource(id, channel, …, trigLevel2)
AcqrsD1_getTrigSource(id, &delayTime, …, &trigLevel2)
```

Discussion

For channels, the second trigger discriminator level can also be controlled relative to the current input range (as defined by the full scale and offset). This is done with **trRelThres2**, which takes values in %FS, and where 0.0 corresponds to the middle of the input range.

trThresUnit

Purpose

Control whether the relative or the absolute trigger discriminator level attributes should be conserved when changing the input range (fullscale and/or offset)

Prefixes	Access	Type	Values
channel & ext	RW	double	"V", "%FS"

Models

DC271-FAMILY

Traditional API equivalent

NONE

Discussion

When the value “V” is set for this attribute, changing the input range of a channel (i.e. changing the full scale and/or the offset) will keep the value in volts set for the trThres1/2 levels constant, and adjust the corresponding relative values, trRelThres1/2.

On the other hand, if the value “%FS” is chosen, a change in input range will imply a change in the value in volts set for the trThres1/2 levels.

trTVField

Purpose

Control the field choice for the TV trigger

Prefixes	Access	Type	Values
ext	RW	string	"Odd", "Even", …

Models

12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configTrigTV (id, channel, …, field, …)
AcqrsD1_getTrigTV(id, channel, …, &field, …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

trTVLine

Purpose

Control the line choice for the TV trigger

Prefixes	Access	Type	Values
ext	RW	long	

Models

12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configTrigTV (id, channel, …, line)
AcqrsD1_getTrigTV(id, channel, …, &line)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

trTVStd

Purpose

Control the standard to be used for the TV trigger

Prefixes	Access	Type	Values
ext	RW	string	"650-50Hz", "525-60Hz"

Models

12-bit-FAMILY

Traditional API equivalent

```
AcqrsD1_configTrigTV (id, channel, standard, …)
AcqrsD1_getTrigTV(id, channel, &standard, …)
```

Discussion

Studying the current range of the attribute may be needed to understand all of the possible values.

5. Appendix A: Estimating Data Transfer Times

The time to transfer a waveform may be a significant part of the execution time of a program, and thus becomes an important design consideration for new applications. We present here a simple timing model with the aim of predicting the transfer time for digitizer readout as a function of the number of segments, number of samples, CPU speed and the operating system.

5.1. Principles & Formulas

The function **AcqrsD1_readData** for readMode = 0 executes a direct waveform transfer from the digitizer memory to the user-allocated buffer, for a single segment at a time. Since direct-to-memory access (DMA) is used, this is the most time-efficient method for segments of 10'000 or more samples. However, each segment requires its own DMA setup. When segments are very short, the transfer overhead starts to dominate the overall transfer time. After the transfer the data in the buffer is ready to be used.

The function **AcqrsD1_readData** for readMode = 1 reduces the overhead time by transferring with a single DMA a complete digitizer memory image to the host computer memory. The memory image is composed of a number of segments, each of which is a *circular* buffer. The first data point of interest in the circular buffer may be anywhere, and its position usually changes randomly between acquisitions. Thus, after the DMA is terminated, the driver must copy data from the image to the final linear buffer for each segment. This method is therefore only interesting for relatively short segments. However, if timing is not an important consideration, it offers the convenience of a single function call for the complete transfer of a waveform sequence.

Formulas for estimating the transfer times in μs are:

- **AcqrsD1_readData** for readMode = 0**:**

$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr$$

- **AcqrsD1_readData** for readMode = 1**:**

$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$

with the following definitions:

M	Number of segments
N	Number of samples per segment
Xfr	Transfer time per sample in μs, typically 0.01 ??
$Ovhd_{DMA}$	DMA overhead time (per segment) in μs
$Ovhd_{buffer}$	Circular buffer analysis overhead time (per segment) in μs
$Extra$	Number of 'overhead' data points per segment
Cpy	Time to copy a sample in μs

The formulas above assume that *M* and *N* correspond to the number segments and samples set with the function **AcqrsD1_configMemory**. If fewer segments or samples are transferred, the timing might be somewhat less favorable than estimated by the formula.

1. The transfer time X_{fr} is typically 0.009 to 0.01 μs (9 - 10 ns) per 8-bit sample for digitizers directly inserted on the PCI bus of the host computer or connected through the SBS-Bit3 interface (Acqiris model number IC200). If a digitizer is connected through the National Instruments MXI-3 interface, X_{fr} is typically 0.012 μs (12 ns). Of course, these times would get larger, if there was considerable additional I/O traffic on the PCI bus.

2. The DMA overhead time $Ovhd_{DMA}$ is approximately (50'000 μsec) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium $Ovhd_{DMA}$ is ~ 200 μs, and for a 500 MHz Pentium it is ~ 100 μs. With applications running under Windows 95 or 98, $Ovhd_{DMA}$ has been observed to be ~ 20% lower.

3. The circular buffer analysis overhead time $Ovhd_{buffer}$ is ~ (2'500 µs) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium $Ovhd_{buffer}$ is ~ 10 µs, and for a 500 MHz Pentium ~ 5 µs.

4. An *Extra* number of data samples must be transferred to the host computer with the memory image. It depends on the sampling interval (in ns) and on the digitizer type. Here are some rough values:

Extra	Type
300 / sampInterval + 98	digitizers with ≤ 1 GS/s maximum sampling rates
300 / sampInterval + 194	digitizers with 2 GS/s maximum sampling rates or DC271-FAMILY instruments set for a combine channel mode allowing 2 GS/s
300 / sampInterval + 386	DC271-FAMILY instruments set for a 4 GS/s combine channel mode

5. The copying time *Cpy* per 8-bit sample is ~ (5 µs) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium *Cpy* is ~ 0.02 µs, and for a 500 MHz Pentium it is ~ 0.01 µs.

Benchmarks were run on 266 and 550 MHz Pentiums, running under Windows 98 and NT. The observed transfer times agreed with the formula within better than 20%.

- **dataType=3**:

Add the conversion time T_3 for conversion from ADC codes to Volts, to T_2 or T_1 respectively:

$$T_3 = M \cdot N \cdot Conv$$

where *Conv* is the conversion time per sample in microseconds. It is ~ (35 µs) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium *Conv* is ~ 0.14 µs, and for a 500 MHz Pentium ~ 0.07 µs.

5.2. Examples

(A) DP210 in a 800 MHz Pentium:

Transferring single records of 100'000 samples, recorded at 2 GS/s:

$$M = 1 \quad N = 100\,000 \quad Xfr = 0.01 \quad Cpy = 0.00625$$
$$Ovhd_{DMA} = 62.5 \quad Ovhd_{buffer} = 3.125 \quad Extra = 480 + 192 = 672$$
$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 62.5 + 1000 = \underline{1063\mu s}$$
$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$
$$= 62.5 + 3.125 + 100\,672 \cdot 0.01 + 100\,000 \cdot 0.00625 = \underline{1698\mu s}$$

It is therefore more favorable to use the function **AcqrsD1_readData** for readMode = 0.

Transferring 100 segments of 1000 samples, recorded at 2 GS/s:

$$M = 100 \quad N = 1000 \quad Xfr = 0.01 \quad Cpy = 0.00625$$
$$Ovhd_{DMA} = 62.5 \quad Ovhd_{buffer} = 3.125 \quad Extra = 480 + 192 = 672$$
$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100 \cdot 62.5 + 100 \cdot 1000 \cdot 0.01 = \underline{7250\mu s}$$
$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$
$$= 62.5 + 100 \cdot 3.125 + 100 \cdot 1672 \cdot 0.01 + 100 \cdot 1000 \cdot 0.00625 = \underline{2672\mu s}$$

The function **AcqrsD1_readData** for readMode = 1 is about 2.7 times faster.

(B) DC270 connected to a 500 MHz Pentium with a MXI-3 interface:

Transferring single records of 100'000 samples, recorded at 100 MS/s:

$M = 1 \quad N = 100\,000 \quad Xfr = 0.012 \quad Cpy = 0.01$

$Ovhd_{DMA} = 100 \quad Ovhd_{buffer} = 5 \quad Extra = 240/10 + 96 = 120$

$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100 + 1200 = \underline{1300\,\mu s}$

$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$

$= 100 + 5 + 100120 \cdot 0.012 + 100\,000 \cdot 0.01 = \underline{2306\,\mu s}$

It is therefore more favorable to use the function **AcqrsD1_readData** for readMode = 0.

Transferring 1000 segments of 500 samples each, recorded at 500 MS/s:

$M = 1000 \quad N = 500 \quad Xfr = 0.012 \quad Cpy = 0.01$

$Ovhd_{DMA} = 100 \quad Ovhd_{buffer} = 5 \quad Extra = 240/2 + 96 = 216$

$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100\,000 + 6\,000 = \underline{106\,000\,\mu s}$

$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$

$= 100 + 1000 \cdot 5 + 1000 \cdot 716 \cdot 0.012 + 1000 \cdot 500 \cdot 0.01 = \underline{18\,692\,\mu s}$

The function **AcqrsD1_readData** for readMode = 1 is about 5 times faster.

5.3. Comparison Chart for Typical Transfers

Time in ms		250 MHz Pentium			500 MHz Pentium			800 MHz Pentium		
# of Segments	*Samples/ Segment*	T_1	T_2	*R*	T_1	T_2	*R*	T_1	T_2	*R*
1	200	0.20	0.22	**1.10**	0.10	0.12	**1.13**	0.06	0.08	**1.17**
1	1 K	0.21	0.25	**1.17**	0.11	0.13	**1.20**	0.07	0.09	**1.22**
1	10 K	0.30	0.52	**1.72**	0.20	0.31	**1.56**	0.16	0.23	**1.45**
1	100 K	1.20	3.22	**2.68**	1.10	2.11	**1.92**	1.06	1.70	**1.60**
1	1 M	10.2	30.2	**2.96**	10.1	20.1	**1.99**	10.0	16.3	**1.62**
10	200	2.02	0.43	**0.21**	1.02	0.26	**0.25**	0.65	0.19	**0.30**
10	1 K	2.10	0.67	**0.32**	1.10	0.42	**0.38**	0.73	0.32	**0.45**
10	10 K	3.00	3.37	**1.12**	2.00	2.22	**1.11**	1.63	1.79	**1.10**
10	100 K	12.0	30.4	**2.53**	11.0	20.2	**1.84**	10.6	16.4	**1.54**
10	1 M	102	300	**2.94**	101	200	**1.98**	101	163	**1.62**
100	200	20.2	2.47	**0.12**	10.2	1.67	**0.16**	6.45	1.37	**0.21**
100	1 K	21.0	4.87	**0.23**	11.0	3.27	**0.30**	7.25	2.67	**0.37**
100	10 K	30.0	31.9	**1.06**	20.0	21.3	**1.06**	16.2	17.3	**1.06**
100	100 K	120	302	**2.52**	110	202	**1.83**	106	164	**1.54**
1000	200	202	22.9	**0.11**	102	15.8	**0.16**	64.5	13.2	**0.20**
1000	1 K	210	46.9	**0.22**	110	31.8	**0.29**	72.5	26.2	**0.36**
1000	10 K	300	317	**1.06**	200	212	**1.06**	163	170	**1.06**
8000	200	1616	182	**0.11**	816	126	**0.15**	516	105	**0.20**
8000	1 K	1680	374	**0.22**	880	254	**0.29**	580	209	**0.36**
8000	2 K	1760	614	**0.35**	960	414	**0.43**	660	339	**0.51**

Comments:

- We assume a 2 GS/s digitizer running at the highest sampling rate, direct connection of the digitizer to the host PCI bus or through the SBS-Bit3 interface (Acqiris model number IC200), Windows NT.
- $R = T_2 / T_1$
  If R > 1.0, a loop over **AcqrsD1_readData** for readMode = 0 is faster than **AcqrsD1_readData** for readMode = 1.

Time in ms		**866 MHz Pentium**		
# of Segments	***Samples/ Segment***	***T_1***	***T_2***	***R***
1	200	0.03	0.04	**1.22**
1	1 K	0.04	0.05	**1.21**
1	10 K	0.12	0.17	**1.45**
1	100 K	0.98	2.12	**2.15**
1	1 M	9.85	25.5	**2.59**
10	200	0.3	0.14	**0.46**
10	1 K	0.35	0.27	**0.76**
10	10 K	1.21	1.61	**1.33**
10	100 K	11.6	16.2	**1.40**
10	500 k	45.4	94.4	**2.08**
100	200	3.00	0.92	**0.31**
100	1 K	4.27	2.37	**0.56**
100	10 K	16.1	15.0	**0.93**
100	50 K	44.7	74.2	**1.66**
1000	200	47.2	13.5	**0.29**
1000	1 K	44.3	23.4	**0.53**
1000	5 K	85.9	87.8	**1.02**
8000	200	457	64.8	**0.14**
8000	500	496	202	**0.41**

Comments: Measured under the Linux OS.